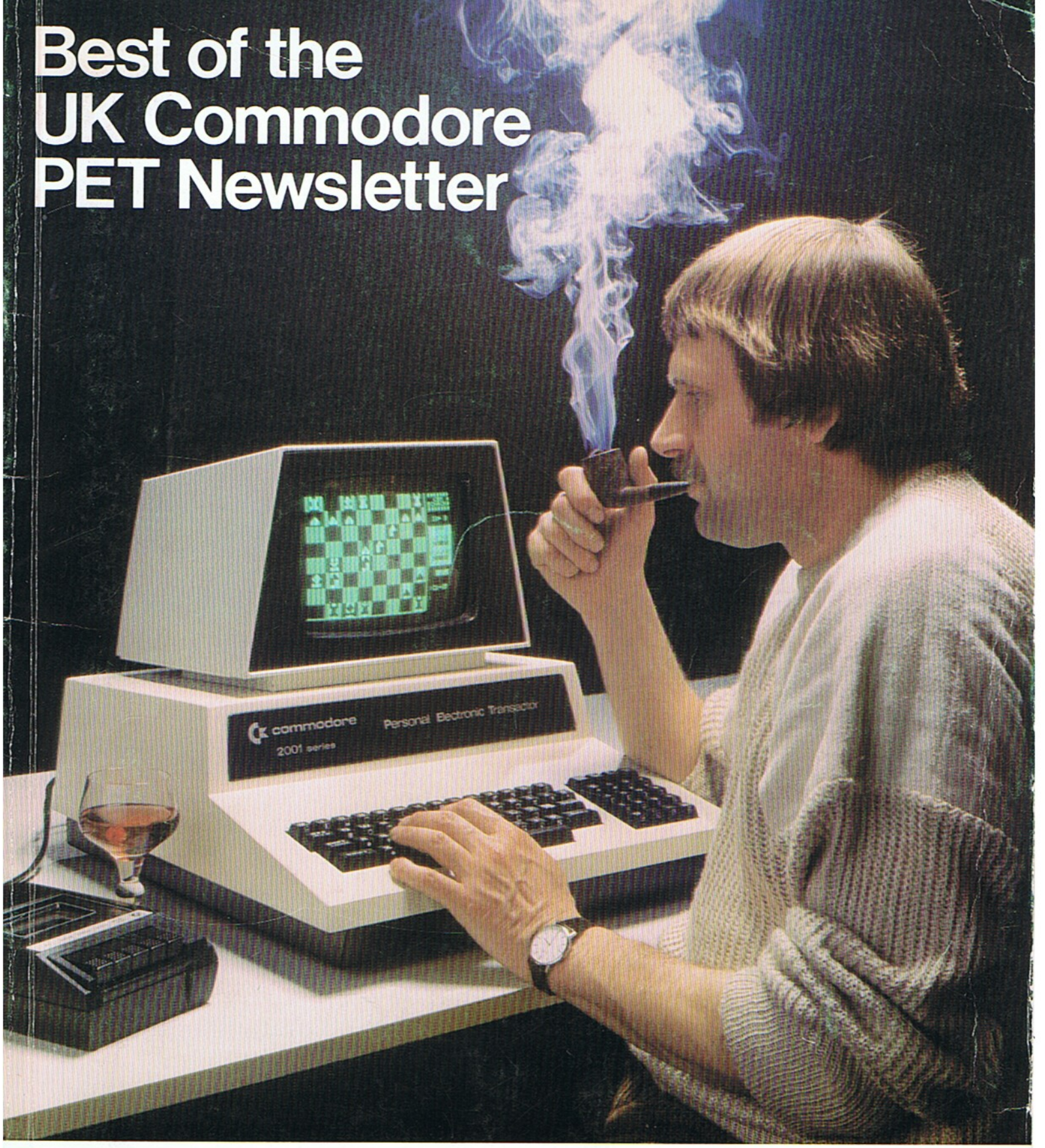
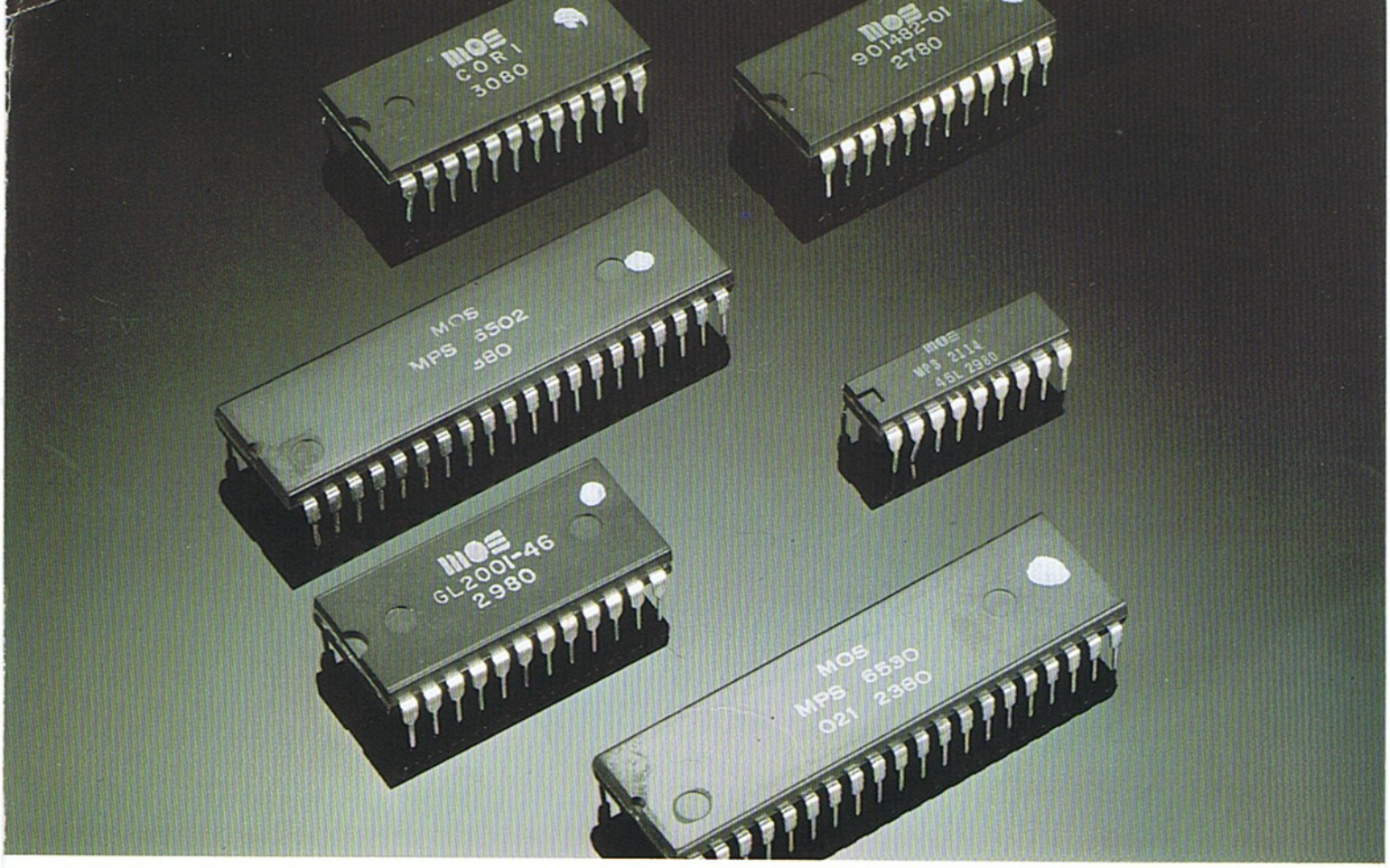
Best of the UK Commodore PET Newsletter



A Compilation of Volumes 1 & 2

Edited by: David Middleton

commodore



From Chips . . .

. . . To Systems

Commodore produces everything

The Best of Commodore PET Newsletter

A Compilation of the Articles from the First Two Volumes of the Commodore PET User Club Newsletter

Edited by: David Middleton

Introduction

It is hard to believe that the Commodore PET has only been in existence for 3 years. The arrival of the PET filled an area in my life which I had only been able to fulfill by using cumbersome main-frame computers at two o'clock in the morning! Now I work for Commodore I can use a PET all day. Unfortunatly I now spend every day sitting at the PET typing and editing articles for CPUCN, I have no time for programming. So I am back to sitting up until 2 o'clock in the morning to fulfil my programming addiction, such is life!

This book is a compilation of all the best articles which have been published in CPUCN over the past two years. All the articles have been rewritten and edited where necessary so that BASIC1 and BASIC2 PETs are covered. If I have missed a conversion for some reason then you will find the tables at the back of the book useful. I have also included the memory maps for BASIC4. BASIC4 is the latest version of the language which is used within the 8000 series 80 column PET and available as a retrofit for the 3000 series machines. BASIC4 is not covered in this book but you should find no problems in converting any of the programs to run on the SuperPET as the main changes are concerned with garbage string collection and advanced disk handing commands.

I hope you will be able to find your way around the book, I have divided it into what I consider to be logical sections. All the bits that were left over have been put into the General section towards the back of the book. I have written a comprehensive alphabetical index at the back of the book and a numeric contents index for the front, if you can not find what you want with either of them then I would suggest the age old technique of random access!

The book is not supposed to be a teaching aid for any specific task, you will not for instance learn how to program in BASIC but you should pick up quite a few tips on how to improve your programs. The same can be said for each of the sections, there is however a good section called Beginning Machine Code which should help beginners to start the strange rites of machine code programming. If you do not learn anything from the book then you should be writing articles for CPUCN pointing out your own discoveries, I expect to hear from you.

Thanks to Andrew Goltz and Helen Elsam for proof reading and to Bob Washington for setting and printing the book.

Contents

MACHINE CODE PROGRAMMING

IEEE BUS

GENERAL INFORMATION

MEMORY MAPS

The Commodore Pet Users Club

WHAT IS THE PET USERS CLUB?

The PET Users Club was founded in order to distribute news, ideas, applications and programs relating to the PET between Commodore and PET Users and to act as an information exchange amongst PET Users. Also, once a year, Commodore stages the PET show. This consists of the latest products produced by dealers and software houses, seminars on PET related topics and a chance to meet some of the people who work at Commodore. The PET show is free to members.

HOW IS THIS INFORMATION DISTRIBUTED?

Eight times a year 'The Commodore PET Users Club Newsletter' or CPUCN is sent out to all our members. This contains product news, programs in BASIC and machine code, tips on programming, reviews of PET compatible hardware and software available from other manufacturers.

CAN I SELL MY PROGRAM THROUGH COMMODORE?

Yes. Many of our program authors are members and rely on the programming hints published in the User Club Newsletter to "tune" their programs to provide maximum performance.

WOULD OTHER MEMBERS BE INTERESTED IN THE APPLICATIONS THAT I HAVE DEVELOPED FOR THE PET?

Most certainly YES. Details of problems encountered and solved by other users are one of the chief benefits of being a User Club member. In order to encourage bashful writers the Commodore PET Users Club give a £50.00 voucher (exchangeable for any Commodore Product) to the author of the best applications article or program published in CPUCN. In addition all articles published are eligible for entry in the competition "Best of the Year" with a £250.00 Commodore Voucher as the prize.

HOW MUCH DOES IT COST?

The subscription fee for the User Club is £10.00 for the UK and £15.00/year for overseas and you get free entry to the Commodore PET Show held once a year in London. The cost of membership will soon be repaid in terms of product information, programs and programming tips.

SPECIAL OFFER TO NEW PET OWNERS!

As a special incentive to new PET owners the membership fee has been reduced to only £5.00 for UK membership and £10.00 for overseas membership but you have to send us a photocopy of your purchase invoice and it must be received by us before the 90 day warrenty period has expired.

CAN I BACKDATE MY SUBSCRIPTION TO INCLUDE PREVIOUS NEWSLETTERS?

There is no point in backdating your subscription. By purchasing 'Best of CPUCN' you are getting all best articles printed in the first two volumes of CPUCN in one easily accessed package.

HOW DO I APPLY FOR MEMBERSHIP?

Simply send a cheque/P.O for £10.00 for within the UK or £15.00 if you live overseas. Remeber you get a £5.00 discount if you send a copy of your purchase invoice within 90 days of purchase. Any appropriate backdated issues will be sent immediately and thereafter you will receive subsequent issues at regular intervals.

Send your cheque to:-

CPUCN Subscriptions
Commodore Business Machines
818 Leigh Rd.
Slough
Berks

MEMBERSHIP FORM

Name_____

Address_____

_____

_____

_____

_____

_____

I enclose a cheque to cover the following items:-

Membership fee (UK) £10.00_____
Membership fee (Overseas) £15.00_____

I enclose a cheque and a copy of my purchase invoice thus claiming my £5.00 discount to cover the following items:-

Membership fee (UK) £5.00_____
Membership fee (Overseas) £10.00_____

Total enclosed £_____

A History of Commodore

Commodore was founded in 1958 by company president, Jack Tramiel, and started life as a Canadian service company specialising in the repair of typewriters. Within two years Commodore had widened the scope of its activities and become a manufacturer of typewriters and a distributor of mechanical calculators.

By 1968 Commodore had sold off its interests in electromechanical calculators and adding machines, and became one of the first companies to market electronic calculators in the United States. The first machines were very large, heavy and capable of only performing simple arithmetic, addition, subtraction, division and multiplication. But gradually calculators started to shrink in size as the technology for fitting more and more components onto the chips increased. In 1971, Commodore was the first company in the world to mass-market a compact personal electronic calculator. In the same year Commodore introduced the first portable rechargeable calculator. Then in 1973 Commodore became a manufacturer as well as a distributor of electronic calculators.

By 1975 Commodore was one of the most successful calculator manufacturers and entered the new digital watch market.

In 1976, as part of a deliberate strategy of vertical integration, Commodore took over Optical Diodes Inc, Frontier Inc, and also significantly MOS Technology, who manufactures the 6500 series microprocessors. MOS also make the KIM, the evaluation board for the 6502 microprocessor. Commodore had entered the computer field.

In 1977 very few people had any idea as to what a personal computer should consist of. The evaluation boards were selling well but they where difficult to use and needed a lot of thought to get even the simplest program working. A simpler more user friendly approach was needed. Chuck Peddle, a senior electronics engineer with Commodore had a few ideas in this area and set about designing a system which would appeal to a mass market.

The PET was officially unveiled at the Consumer Electronics show in Chicago. Sitting amongst the calculators and watches on the Commodore stand was an 8k PET. The PET caused a sensation at the show and put the evaluation board market out of business almost overnight, this was what people really wanted. Initially Commodore quoted 30 days delivery but demand was so high that after a while the quoted delivery time was 4-5 months and this was on a pre-paid order!

After a year during which the 8k PET firmly established itself in the market Commodore announced the arival of the 3000 series large keyboard machines aimed at the potentially enormous business market. This new PET with a large standard keyboard and an upgraded operating system capable of accessing a disk system turned the PET into a machine capable of taking on the giants.

In February 1979 Commodore consolidated production of the PET so that everything took place under 1 roof, rather than 3 seperate locations in Palo Alto, this being at the purpose built solar powered 60,000 sq foot facility in Santa Clara.

With the market now becomming more competitive Commodore had to keep the pressure on to stop new manufactures squeezing the PET out of potential markets, so research was undertaken within Commodore to accommodate the increasing needs of the business market. The CBM 8000 series, dubbed the SuperPET, consisting of the 80 column screen PET and a 1 Megabyte disk system, arrived in July 1980 with the obvious intention of entering the word processing field. A system which can compete with IBM giving more power for a third of the cost is going to cause considerable turmoil in the early 80's.

What the future is going to bring is very difficult to predict for more than about 6 months because developement is accelerating all the time. Small colour PETs, PETs with larger memory capacity and multi-megabyte disk systems are all being designed to complement the current range and will continue to give the best value for money whilst giving excellent dealer backup.

Book Reviews

UNDERSTANDING YOUR PET/CBM
Dave Middleton

Vol.1 - Basic Programming
By: D.Schultz and D.Smith
Cost: 15 pounds
Available from: PETSOFT

The book is 240 pages long and is packed with information which the absolute beginner will find invaluable. It would appear that the book is mainly a compilation of the popular TIS workbooks with the bemused lion making explanatory appearances throughout. The text starts off each section with very simple examples and then builds on these slowly to show more complex methods.

The sections cover just about everything needed to get a minimum system (ie. PET + cassette deck) into operation. Unfortunatly the amount written about using a disk system with the PET can be held on one page which is a bit of a shame when 46 pages are devoted to the cassette. Possibly the next volume, which is not available yet, will contain disk programming.

If anything the book takes teaching by example too far and the user may become bored plodding through every program but this is something which is an asset rather than hindrance as is means that nothing is assumed by the authors and if the reader wishes to skip a few pages then it is his own fault if he does not understand later. This way of presenting material is something which a lot of text books could take example from, too much information is far better than too little.

PET AND THE IEEE 488 BUS (GPIB)
Dave Briggs

By: Eugene Fisher-CW Jensen.
Publishers: McGraw-Hill
Cost: 9.95 pounds
Available from: Audiogenics

If like me, you have often used the PET IEEE Bus and consider that you have a fair idea of how it works then this book could come as quite an eye opener.

With prolific use of tables and timing diagrams the first six chapters give a detailed and concise description of how the PET IEEE Bus operates for the various combinations of commands that are available. Non-standard interfacing and the differences between the PET IEEE and the standard GPIB are also covered in the same lucid style.

A chapter on specialised applications is the only one that goes into more depth and technicalities than the first time user will require.

The book finishes with a chapter on how to find out what is wrong when the Bus is apparently not working correctly and a series of appendices, including a very comprehensive bibliography, add an equally comprehensive listing of devices with GPIB interfaces.

All in all the book gives an excellent insight into the workings of the PET IEEE 488 Bus which will be of value to both first time and more advanced users. It should definitely appear in the library of all serious PET users.

PROGRAMMING THE 6502
Andrew Goltz

by: Rodney Zaks
publishers: SYBEX
Available from: Commodore

For many years our own MOS Technology Hardware and Programming Manuals have been held in high regard by digital engineers and programmers working with the 65xx family of microprocessor and peripheral chips. However, when these manuals first appeared, the basic assumption of the publisher was that their readership would be confined to those already familiar with the fundamental concepts of microprocessor design and implementation.

The massive new interest in microprocessors means that the rules have changed somewhat during the half-decade that has passed since MOS Manuals first appeared. The launching of the hardware development board: KIM, followed by PET itself introduced a new type of user anxious to learn to program the 6502 microprocessor at machine code level. Many PET Users, who starting with no previous computer experience, have successfully used PET as a tool for teaching themselves programming in BASIC, approach me with requests for an equivalent of "Basic BASIC" for learning machine code.

Until now I have been recommending Volume One of the classic trilogy: "An Introduction to Microcomputers", by Adam

Osborne as providing the best tutorial available on the fundamental concepts prior to tackling the MOS manuals. But although Osborne's book is excellent in its exposition of the general principles, there is clearly a demand by users of systems based on the 6502 for an introductory level text dealing specifically with this microprocessor. Now this missing book has been written by Rodney Zaks, and is available in the U.K.

"Programming the 6502" assumes no previous technical background, and takes the reader, by step, through the jungle of computer jargon. Basic concepts such as:- creation of an algorithm, flowcharting, binary representation, octal and hexadecimal notations are clearly explained. The same thorough approach continues with sections of the book dealing with 6502 hardware organisation and basic programming techniques - guiding the reader to the point where he can move on to reading the MOS Hardware and Programming manuals with no further assistance.

A special section is devoted to input/output techniques which will be invaluable for anyone intending to use their PET to monitor scientific equipment, or simply run their home or office security system.

Another section gives and explains typical machine code application programs such as getting a character from a keyboard, clearing a portion of memory and finding the largest element of a table (machine code subroutines for sorting can slice minutes off from the execution of BASIC programs).

The concluding chapters of the book look at the general principles of data structure organisation and program development and serve as an excellent introduction to the more specific books that are available.

BASIC COMPUTER GAMES

By: David H. Ahl (Editor)
Cost: 4.50 pounds
Available from:
Transatlantic Book Service Ltd.
24 Red Lion Street
London WC1R 4PX

As the title suggests, this is a collection of computer games written in BASIC. The games come from 'Creative Computing' and have been collected and adapted largely by David Ahl (founder and publisher of C.C.). As all the programs are written in Microsoft 8K BASIC they are completely compatible with the PET and the listings can be entered into the keyboard and recorded for later use.

The book contains 101 games in total, ranging from old favourites including Lunar Landing and various card games, to some refreshingly original games such as Animal, and Poetry. On the whole, the games tend to be of the static type, requiring mental judgement rather than just fast reactions. The listings and documentation are presented in a tidy manner and on the whole, are unambiguous. A large number of anthropomorphic computer cartoons gives the publication an informal and 'friendly' image.

On the negative side, most of the listings could have been made considerably more efficient without loss of clarity. I found, however, that the programs were relatively easy to compress when entering them into the PET - saving time and memory space. Also, PET's graphic capabilities enable several of the games to be made more visually interesting with a little extra work. Having said that, the book represents very good value for money.

6502 APPLICATIONS BOOK

Richard Pawson
By: Rodney Zaks
Publishers: SYBEX
Available from: Local dealers

This book starts where "Programming the 6502" left off. Rodney Zaks is a well respected author of computing publications and has had considerable experience in training newcomers to the field.

The "6502 Applications Book" deals with all the Input/Output associated with 6502 based systems. As such, the book is particularly relevant to PET users who wish to interface peripherals and 'home grown' devices to their computer.

A large amount of space at the beginning of the book is devoted to explaining the intricacies of the various support chips manufactured for the 6500 range of microprocessor. These include PIO's such as the 6520 for handling parallel data complete with hand shaking, and the more advanced Versatile Interface Adaptors like the 6522 which contains two programmable timers and a Shift Register for converting between serial and parallel data. The principles and operation of these devices are explained in detail from absolute basics and the text is interspersed with numerous diagrams, and worked examples. In addition there is a large number of test questions enabling you to keep a check on your understanding of each new concept.

The rest of the book deals with the practical side of interfacing and is illustrated with several diverse applications including; a traffic control system, a burglar alarm, a music generator and many others. All the

assembly language routines (an understanding of A.L. programming is a pre-requisite) are written in a machine-independent form, for ease of application.

If you intend to interface any device to your PET, or simply want to learn how a system like the PET works internally, then this is probably one of the best books available on the subject.

The "6502 Applications Book", as with the previous book, is published by SYBEX and should be obtainable from your nearest computer bookshop.

PER/CBM PERSONAL COMPUTER GUIDE

By: Carroll Donahue and Janice Enger
Cost: 9.95 pounds
Publishers: Osborne/McGraw-Hill
Available from: Audiogenic

This latest publication from Adam Osborne/McGraw Hill maintains the excellent standards established by their earlier classics such as 'An Introduction to Microcomputers' and '6502 Assembly Language Programming'.

The book is a well written, comprehensive guide to the PET that will be found useful by the novice, beginning programmer and experienced PET user. The material in the book being well 'signposted' to indicate the level of difficulty.

As expected in Osborne publications diagrams and worked examples are extensively used throughout the book and they successfully complement the main text. A new departure is the use of photographs and screen displays which help to make the book 'friendly' for the absolute beginner.

Sometimes the book appears to be guilty of providing too much information e.g. the section of PET trivia includes a discussion of what happens when three keys on the PETs keyboard are depressed simultaneously! But in most instances the reader is warned that he is reading a section which may be skipped at first and often such sections provide the experienced PET programmer with useful reference material.

The book includes an introductory chapter that explains the basic concepts common to microcomputers and looks at PET's special features. The next chapter called 'Operating the PET' deals thoroughly with the installation, correct use and maintenance of the PET hardware. The next two chapters deal comprehensively with PET BASIC commands and statements and the techniques of writing, editing and saving programs. Chapter 5, entitled 'Making the most of the PET features', discusses string handling, programmed cursor movement, graphics, animation and file handling. PET special features, including the graphics character set, internal clock and Poking screen memory, are well covered. The final chapter will be especially useful for advanced programmers as it deals with the details of PET's own operating system, and introduces the concept of Machine Code programming.

A comprehensive series of appendices provides memory maps and tables for BASIC 1.0 and BASIC 2.0, and an excellent bibliography completes a well thought out and carefully researched book. I have no hesitation in recommending a copy for every PET user's book-shelf.

Sorting Techniques

A FAST SORT
Jim Butterfield

When you need to sort a large array, sorting speed becomes important. Most simple sorts become very slow, since twice as many items will take four times as long to sort.

This fast sort is called "selective replacement"; it's classified as a tree type sort. It needs an index array, called I(J) here, which is twice the size of the items to be sorted. Memory can be saved, if needed, by making it an integer type array.

```
100 DIM I(200),N$(100),A$(100)
110 PRINT"■ FAST SORT DEMONSTRATION"
120 INPUT "HOW MANY ITEMS";N
125 INPUT "1 OR 2 FIELDS";F
130 FORJ=0TON-1
140 INPUT"NAME";N$(J)
150 IFF<>1THENINPUT"ADDRESS";A$(J)
170 NEXTJ
180 PRINT:IFF=2THENINPUT"SORT BY FIELD
    1 OR 2";W:PRINT
200 TI$="000000"
210 X=0:B=N-1:FORJ=0TOB:I(J)=J:NEXTJ
220 FORJ=0TON*2-3STEP2
230 B=B+1:I1=I(J):I2=I(J+1)
240 GOSUB700
250 I(B)=I:NEXTJ
300 X=X-1:C=I(B):IFC<0GOTO800
320 PRINTN$(C),:IFF<>1THENPRINTA$(C)
340 I(C)=X
350 C%=C/2:J=C%*2:C=N+C%:IFC>0GOTO300
370 I1=I(J):I2=I(J+1)
380 IFI1<0THENI=I2:GOTO410
390 IFI2<0THENI=I1:GOTO410
400 GOSUB700
410 I(C)=I:GOTO350
700 IFW=2GOTO720
710 I=I1:IFN$(I2)<N$(I1)THENI=I2
715 RETURN
720 I=I1:IFA$(I2)<A$(I1)THENI=I2
725 RETURN
800 PRINT:PRINT"■ TIME TO SORT ■";N;"IT
    EMS =";TI/60:IFF=2GOTO180
```

THE SHELL—METZNER SORT
Mike Niklaus

Most books on BASIC show how to code a 'ripple' or 'bubble' sort. Those of you who have tried it will know that it does its job accurately but slowly. This is because it 'bulldozes' its way through the problem, rippling one item to the end of the list on each pass. There are a number of sort routines which are faster for most sets of data. One of them is the Shell-Metzner.

Like the Bubble sort it compares and swops elements. It differs in the way it selects elements for comparison. The underlying theory is complex and perhaps the best way of seeing the sort in action is to use a pack of playing cards, pencil and paper and 'dry run' your way through a sample sort.

If you have the patience to do this you will find that the routine compares elements half the array apart, then elements a quarter of the array apart and so on. Even if you haven't the time to work out how it operates there is no reason why you shouldn't use it in your programs. It is a perfect example of the type of routine where you know what goes in, what comes out, the variables involved and who cares how it works!

It is also a good example of how you can improve your BASIC skills be reading other people's programs. I learnt about this sort by extracting it from a program published in KILOBAUD last year.

As set out here it sorts an alphabetic array held in element one onwards, into ascending order. It is coded as a subroutine and is a perfect candidate for the Butterfield/Templeton merge described elsewhere.

```
59000 REM SHELL-METZNER SORT
59005 N=A:M=A:REM 1 TO A IS ARRAY SIZE
59010 M=INT(M/2):IF M=0 THEN RETURN
59020 J=1:K=N-M
59030 I=J
59040 L=I+M
59050 IF A$ (I)<=A$(L) THEN 59080
59060 F$=A$(I):A$(I)=A$(L):A$(L)=F$:I=I
      -M:IFI<1THEN59080
59070 GOTO59040
59080 J=J+1:IFJ>KTHEN59010
59090 GOTO59030
```

Variables used:- A$(),F$,A,I,J,K,L,M,N.

I've not yet tried adapting the routine to deal with element zero. Would someone like to contribute a version? Also I will be delighted to feature other sort routines you let me know about.

QUICKSORT
Mike Gross-Niklaus

Several people have written to me following my re-publication of the SHELL-METZNER sort. All have pointed out

that the QUICKSORT, by Hoare, is even quicker than the SHELL.

Working on the principle that it is easier to sort two small arrays rather than one large one, it chooses an element, and splits the array in two, placing smaller elements earlier and larger elements later in the array than the chosen element. It then operates on each of the smaller arrays in exactly the same way. Rather like a clerk sorting manually into smaller and smaller piles, the routine eventually produces sub-arrays which are in order and contain only one or two elements each. One final exchange in the case of the two element sub-array produces the sorted data.

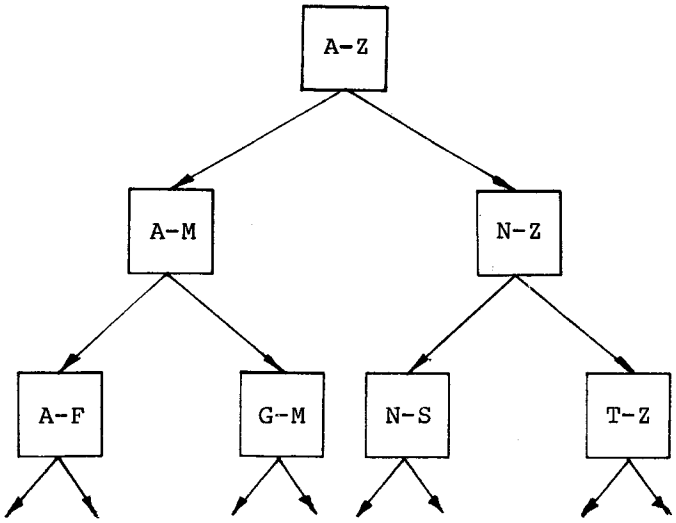
QUICKSORT likes jumbled data. On average the sort time for N elements will be proportional to N*LOG(N). The time for the SHELL will be N to the power of 5/4 and for the Bubble, N squared - on average.

Hoare suggests choosing the first element as the comparison item. However, when the array is ordered or nearly so, most other elements get placed on one side of it, creating a very small and a very large sub-array. In this case sort times approach that of the Bubble sort. One answer suggested by Harrington is to choose the middle array element for comparison. This works fine for ordered arrays but not for two ordered arrays joined end to end. He suggests that choosing an element at random is the way out of this problem.

You will see that where you are looking for the quickest way to sort data, it is necessary to consider the nature of that data and depending on how it is arranged, choose a suitable routine. If you can't predict the nature of the data, then probably the best compromise is the SHELL.

The following listing is taken from Harrington's article in "Micro Computing".

```
95 REM SET ARRAY SIZE
100 LET M=N
105 REM SET SPACING BETWEEN LIST MEMBER
    S
110 LETM=INT(M/2)
115 REM SEE IF DONE
120 IFM=0THEN300
130 LETJ=1
140 LETK=N-M
145 REM I & L INDICATED ELMNTS TO BE ME
    RGED
150 LETI=J
160 LETL=I+M
165 REM PERFORM MERGE
170 IFA(I)<A(L)GOTO240
180 LETT=A(I)
190 LETA(I)=A(L)
200 LETA(L)=T
210 LETI=I-M
220 IFI<1THEN240
230 GOTO160
235 REM MERGE NEXT TWO ELEMENTS
240 LETJ=J+1
250 IFJ<=KTHEN150
255 REM BEGIN MERGE OF RESULTANT LISTS
260 GOTO110
300 REM END OF SORT
```



SORTING BY INSERTION AND CHAINING

L. J. Slow, Bradford Grammar School

This method requires only three passes through the array (A), together with some passing along short chains; thus the sort-time is approximately linearly dependent upon the number of elements (N+1).

The value of each element in A is transformed into an integer (E) in the range Ø to N, inclusive. This integer depends on the relative size of the value in A and is the position in the array B% where the element number of the current value should be stored. However, if this position is already occupied then a supplementary array C% is used to begin a chain of element numbers corresponding to those values in A that give rise to the same E. Each link in the chain holds the element number of the next link; the last link holds the value -1. Links in each chain are positioned so that the corresponding values in A are in ascending order. At the end of the INSERTION and CHAINING loop, B% will contain the first links in each chain with -1's remaining where no chains begin, and C% will contain all the chains interleaved with each other and residual -1's.

This method is at its most efficient when the values to be sorted are in a uniform distribution. However, if the values

follow a normal distribution, a function that maps this into a uniform distribution could be considered, but longer calculation time could offset the advantage on a 32k machine.

Variable dictionary

A Array to be sorted.
B% Array of first links and chains.
B A particular element of B%.
C% Array of all other chain links.
C A particular element of C%.
D Duplicate array of A.
E Transformed element of A.
F Mapping factor.
I,J Counts.
MAX Maximum value of A.
MIN Minimum value of A.

All arrays have N+1 elements.

LINE NUMBER EXPLANATION

50220
The current element number is inserted at the point E in the array B%, unless a chain already starts there.

50230
If a chain has been started it then has to decide of the first link in the chain should be displaced. This is done if the current value of A is less than that corresponding to the first link, in which case the first link becomes the second in the chain.

50240
However, if the first link is in the correct position, control passes to the next link. If there are no more links, the number is placed at the end of the chain.

50250
If there are more links, reference is made to the array A to decide whether the current element number should displace the next link, this being done if appropriate.

50260
If the current element number is still unplaced, control passes further along the chain.

Example dry run

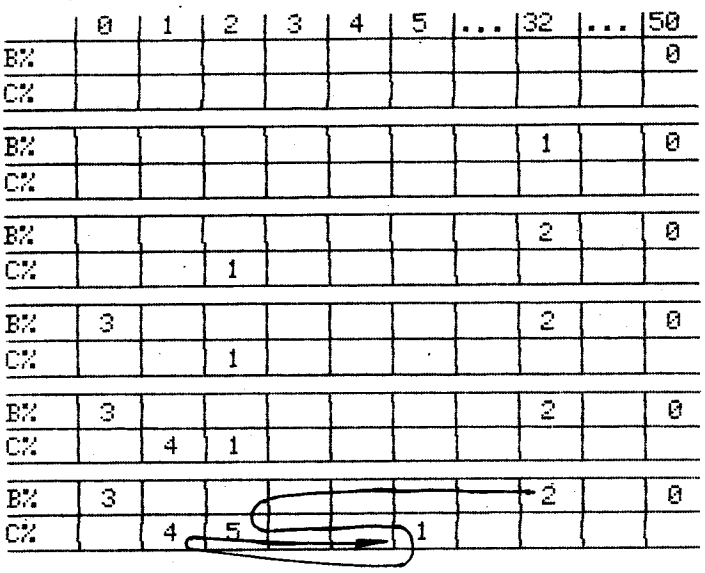
Suppose the the array has 51 elements (N=50), with MAX=70.0, MIN=-30.0 and F=0.5.
If the first six values in A are:-

```
70.0, 35.5, 35.1, -30.0, 35.8, 35.6,...
  0     1     2      3     4     5
```

then the corresponding E's are:-

```
 50    32    32     0    32    32
```

The following table shows how these values of E are used to insert and chain element numbers in the arrays B% and C% step by step. For the sake of clarity, -1's are ommitted.



Note how the element numbers are used to point to the next link in the chain:-

2 to 5 to 1 to 4, where 4 is the current end of this chain. (35.1 to 35.3 to 35.5 to 35.8)

```
10 REM  TEST PROGRAM FOR SUBROUTINE.
20 PRINT"HOW MANY NUMBERS";:INPUT M:
   PRINT
30 IF(M<>INT(M))OR(M<2)GOTO 20
40 N=M-1:IF N<256 GOTO 70
50 PRINT"**WARNING,FOR OLD ROMS THE**"
60 PRINT"********LIMIT IS 256*********"
70 DIM A(N)
80 FOR I=0 TO N
90 A(I)=(RND(TI)-.5)*1000:PRINTA(I)
100 NEXT
110 T=TI
120 GOSUB 50000
130 T=TI-T:T=INT(T*5/3+.5)/100
140 FOR I=1 TO N
150 IF A(I)>=A(I-1) GOTO 170
160 PRINT"ERROR":STOP
170 NEXT:PRINT
180 PRINT"FOR";M;"NUMBERS,TIME=";
190 PRINT T;"SECS."
200 PRINT"FREE MEMORY=";FRE(0);"BYTES."

205 PRINT:FORI=0TON:PRINTA(I):NEXT
210 END
50000 REM  SUBROUTINE TO PLACE THE ELEM
      ENTS OF A GIVEN ARRAY
50010 REM  IN ASCENDING ORDER BY INSERT
      ION AND CHAINING
50020 REM
50030 REM  INPUT PARAMETERS: N & A(0),.
      ..,A(N)
50040 REM
50050 DIM B%(N),C%(N),D(N)
50070 REM  SET INITIAL VALUES AND FIND
      THE MAXIMUM AND MINIMUM VALUES OF
       A.
50080 MAX=A(0):MIN=A(0)
50090 FOR I=0 TO N
50100 B%(I)=-1:C%(I)=-1:D(I)=A(I)
50110 IF A(I)>MAX THEN MAX=A(I):GOTO 50
      130
50120 IF A(I)<MIN THEN MIN=A(I)
50130 NEXT
```

```
50140 IF MAX=MIN GOTO 50360
50150 REM
50160 REM  CALCULATE MAPPING FACTOR.
50170 F=INT(N*1000/(MAX-MIN))/1000
50180 REM
50190 REM  INSERTION AND CHAINING LOOP.

50200 FOR I=0 TO N
50210 E=INT((A(I)-MIN)*F):B=B%(E)
50220 IF B=-1 THEN B%(E)=I:GOTO 50270
50230 IF A(B)>A(I) THEN B%(E)=I:C%(I)=B
      :GOTO 50270
50240 C=C%(B):IF C=-1 THEN C%(B)=I:
      GOTO 50270
50250 IF A(C)>A(I) THEN C%(B)=I:C%(I)=C
      :GOTO 50270
50260 B=C:GOTO 50240
50270 NEXT
50280 REM
50290 REM  UNPACK CHAINS AND PUT ARRAY
      IN ORDER
50300 J=-1
50310 FOR I=0 TO N:B=B%(I)
50320 IF B=-1 GOTO 50350
50330 J=J+1:A(J)=D(B)
50340 B=C%(B):GOTO 50320
50350 NEXT
50360 RETURN
```

The Disk System

The addition of the disk system to the range of peripherals for the PET has turned it from being a usable but time consuming machine with cassette storage into a true computer system with enormous potential.

The information given in the next few pages is directed at DOS 1 disk systems ie 2040 and 3040 exclusively. There is very little information on the 8050 disk as yet but as it becomes available it will be given in Volume 3 of CPUCN.

There have been a number of faults with the disk system which have been pointed out in CPUCN over the months but these are fairly minor items can can easily be programmed around once they are known.

Probably the worst error which is going to be encountered regularly is that of not giving a drive number when SAVEing or OPENing a file. This can cause all sorts of havoc in certain circumstances. See the notes:- 'Corrections to Commodore Floppy Disk User Manual (2204Ø & 3Ø4Ø)' for more information.

Another area where many programmers have expressed concern is in writing direct or random access files. The techniques, once mastered are fairly simple to use. The article by Mike Gross-Niklaus has helped a great number of people to get started with direct access.

DISK INITIALISATION

It is most important that the door of the dirve to be initialised should be left open for 1-2 seconds after the initialise command has been sent.

The effect of leaving the door open during the early stages of initialisation is to allow the disk to settle into its optimum position on the drive before any read/writes take place.

DISK BUG

If you write variable length sequential files and read them back with INPUT# and test for End of File with ST then BEWARE! Files 254 bytes long or any integer multiple thereof will cause the PET to hang up if read by INPUT# using the status test. The cause of the bug is an extra carriage return being placed at the end of the file if it is a factor of 254 bytes long. The solution is to write a character without a carriage return, CHR$(13), as the final character in the file,a line feed, CHR$(1Ø), is suggested. INPUT# will still accept the character even when a carraige return is not the last character in the file.

The program to illusrtate the problem with a fix is shown below. Congratulations to Nick Marcopoulos for an elegant solution.

```
 2 REM RUN THIS PROGRAM WITH AND WITHOUT
 4 REM LINE 90
 5 FORX=1TO3: PRINT
 10 OPEN1,8,5,"@0:TEST,S,W"
 30 FOR J=1 TO X
 40 PRINT#1,"X";
 50 NEXT
 60 FOR I=1 TO 126
 70 PRINT#1,"X";CHR$(13);
 80 NEXT
 90 PRINT#1,CHR$(10)
 100 CLOSE1
 110 OPEN1,8,5,"0:TEST,S,R"
 120 INPUT#1,A$: SS=ST
 130 PRINTA$
 140 IF SS AND 64 THEN CLOSE1: NEXT: END
 150 GOTO120
READY.
```

PROGRAM MERGE FROM DISK

R. I. Wilson of Cortex

The following program very easily allows programs to be appended or merged from a disk file in source code not the saved image form (ie. as it would look when LISTed rather than as it is stored as tokens). This allows programs from other systems to be used without manually typing in the program again. Various other programs already exist to perform this function but they rely on fooling the PET into accepting input from a source other than the keyboard. This program simply opens the required file as normal, inputs and displays a line on the screen and then forces the system to accept that line by pushing carriage returns into the keyboard buffer and ENDing. An immediate mode GOTO is also on the screen causing a return to the program when the line has been accepted (more than one GOTO is displayed in order to catch any errors from READY etc.)
However the system effectively closes all files when any new line is input, by forcing location 174 to 1 the file is now OPEN again and the process is repeated until the End of File marker is reached.

When this point is reached the program deletes itself from the PET and then lists the current contents.

```
63992 INPUT"FILENAME";N$: N$="0:"+N$:
      OPEN1,8,2,N$+",S,R": POKE0,0
63993 PRINT"■■■";
63994 POKE174,1: IF PEEK(0) THENCLOSE1:
       GOTO63998
63995 GET#1,A$: POKE0,ST: PRINTA$;: IF
      A$<>CHR$(13) GOTO63994
63996 PRINT"GOTO63993": POKE158,5: FOR
      I=0 TO 5: POKE623+I,13: NEXT
63997 PRINT"■■■■GOTO63993": PRINT"■";:
      END
63998 PRINT"■■■": FOR I=0 TO 7: PRINT63
      992+I: NEXT: POKE158,10
63999 FOR I=0 TO 9: POKE623+I,13:NEXT:
      PRINT"LIST": PRINT"■": END
```

APENDING PROGRAM FILES FROM DISK

It is very useful to be able to append portions of a program from a subroutine library and even more useful if these can be readily pulled in from disk. Unfortunately, the Append command in Commodore's Disk Utility Maintenance package will only work with data files. In order to append program files it is necessary to get rid of the zeroes which precede the End of File marker and indicate the end of one program and also the starting address which indicates the beginning of the other.

The routine listed below will carry out these functions, and has been successfully used to append the test routine to Mr. Slow's sort program which appears elsewhere.

```
5 E$=CHR$(0)
10 INPUT"1ST SOURCE FILE NAME";A$
20 INPUT"2ND SOURCE FILE NAME";B$
30 OPEN15,8,15,"I"
40 OPEN5,8,5,A$+",P,R"
50 GOSUB1000
60 INPUT"DESTINATION DRIVE#";S$
70 INPUT"DESTINATION FILE NAME";C$
80 OPEN6,8,6,S$+":"+C$+",P,W"
90 GOSUB1000
100 GET#5,D$:IFST<>0GOTO140
110 IFD$=""THEND$=CHR$(0)
120 IF F THENPRINT#6,E$;
130 F=-1: E$=D$: GOTO100
140 CLOSE5
150 OPEN5,8,5,B$+",P,R"
160 GOSUB1000
200 GET#5,A$:GET#5,A$
210 GET#5,D$:T=ST
220 IFD$=""THEND$=CHR$(0)
230 PRINT#6,D$;
240 IFT=0THEN210
250 CLOSE5:CLOSE6:CLOSE15
260 END
1000 INPUT#15,ER,ER$,ET,ES
1010 IFER<>0GOTO1030
1020 RETURN
1030 PRINTER,ER$,ET,ES
1040 STOP
```

CORRECTION TO COMMODORE FLOPPY DISK USER MANUAL (2040 & 3040)

Page 17 & 29
SAVE AND OPEN WITH REPLACE

Reference: SAVE"@dr:fn:",dn
OPEN3,8,5,"@0:JDATA,USR,WRITE

Do not use SAVE with Replace or OPEN with Replace (e.g. SAVE"@0:"FILENAME",8). These commands have a bug which can cause other files to be corrupted on the disk.

Options:

1. Scratch old file first then save new file.
2. - Save new file under a dummy name.
   - Scratch old file.
   - Rename dummy file to correct name.
3. Save file on other drive and use old file as a back-up.

Page 16 and 17
SAVE AND OPEN WRITE FILE WITHOUT DRIVE NO.

The DOS searches for the filename starting on the most recently used drive. If it is not found on either drive as an existing file or as an unclosed file, then the DOS assigns the file entry to the most recently used drive. The contents, however, are placed on the opposite drive.

The result is a file entry which is linked to space on the diskette which may be used by other files or not at all. The BAM of the opposite drive is updated by the DOS.

The contents of the other files on each diskette are not altered, but care should be taken in the recovery procedure. If either drive is empty or un-initialised, the process will halt, leaving the file unclosed.

Recovery Procedure:

The safest way to recover is to properly save the file on another diskette, and copy all files of interest from the diskette with the bad entry to the other diskette. The corrupted diskette may be restored with the verify command.

An easier but less guaranteed method is to scratch the file entry and then restore both disks with the verify command.

Page 19 (and 25)
FORMATTING WITH WRITE PROTECT TAB ON (NEW WITH ID or DUPLICATE)

If an attempt is made to format a disk which has a Write Protect Tab in place, the system will attempt to write. After the system detects that no writing is taking place, an error condition is generated, but the write signal line is

left on. Any subsequent operation will result in writing to the drive involved in the operation.

Since the write protect is also a hardware disable, any protected diskette will not be disturbed. To recover from this situation, power reset the disk unit.

Page 25
DUPLICATE—Write Error

If a write error is encountered in the DUP command, the disk unit will attempt the write continuously. This indicates probable media failure and the diskette should be discarded. This problem may be detected by watching the R/W head or listening carefully to the disk. If the click sound of the head changing tracks is not noticed after 1 minute, then more than likely the problem has occurred.

Page 25
VALIDATE

The manual states that the diskette should be INITIALISED immediately after a Validate error. If this is not done, a loss of file contents will be inevitable, since the BAM in memory is left in an indeterminate state.

Page 29
RENAME—File Not Renamed

This problem has been observed on occasions. Usually adding a file to the diskette will allow RENAME to work. Also, copying the file to the new name will also work.

Page 43
BLOCK ALLOCATE AND BLOCK FREE, Illegal Track or Sector Requested

The B-A & B-F commands will generate an error message which overlays part of the previous message. The position is dependent on the length of the incoming command. For example, if the previous message was 00, OK, 00, 00 and the command was 38 characters long, subsequent inputs would still access the previous message.

Care should be taken when transmitting these two commands. It should be as short as possible and Track & Sector should be legal values. The following table indicates the legal ranges.

Track	Sector
1-17	0-20
18-24	0-19
25-30	0-17
31-35	0-16

Page 43
BUFFER-POINTER AT 0

The B commands are intended for Record oriented disk access. Since position 0 is used as the pointer to the last valid data byte in the record, it is not normally accessed. If it is necessary to access this byte, (for gaining access to a track link for example) B-P at 0 will allow access (GET# or PRINT#) only if the last character pointer is not 255.

Solution:

```
OPEN1,8,15
OPEN2,8,2,"#0"
PRINT#1,"M-R"CHR$(0)CHR$(17)
GET#1,A$:REM 1st Byte of Buffer
```

Page 44
MEMORY—READ GET# without EOI

The Memory-Read command is intended to provide an unlimited access to any part of the file interface controller's memory. The byte read from the memory is placed in the error buffer and the character count is set to one. EOI is not sent with the byte. Consequently, an INPUT# from the Error channel (SA=15) will not terminate. If M-R is to be executed, only a GET# should be used in accessing the byte.

DOS RENAME FAILURE

Harry Broomhall has contributed the following notes about the DOS RENAME failure.

If the following conditions are true, RENAME will not work for any file in the directory :-

1. The last sector in the chain of directory sectors contains either no directory entries or only scratched entries.

2. In any previous sector of the directory there is at least one scratched entry.

Harry has found when both conditions are fulfilled, RENAME will appear to work, giving "OK" as the response down the error channel but, in fact, the RENAME process has not taken place. The only easy cure is to copy the old file providing it with a new name and then scratching the original file.

To recover from this condition, copy all files onto a new disk using the Commodore utility, Copy Disk Files. This will work provided all files are of the non-User type.

RANDOM ACCESS PROGRAMMING

Nick Green

I am still getting requests for more about disk programming. So here are some notes which may be of interest to you, in addition you may find it useful to refer

to a standard work: 'Computer Data-Base Organisation' by James Martin, Prentice-Hall, 1975. Martin's works, although IBM orientated, are always extremely readable. Closer to home, we have the program Random 1.0 listed in the back of the disk manual and distributed to you on the demonstration disk. It may seem rather long, but it shows many of the main features of Random Access Programming. In addition there are our own intensive training courses for those wanting to get rapid "driving lessons" in computing techniques.

Using an internal relative record number, it constructs a descriptor file. This is a sequential file of keys obtained from the first field (sub-record delimited by a carriage return). Note that a key is a distinguished portion of text held in a record used to compute, possibly via a "hashing" algorithm, the physical disk address. This file can then be examined for a particular attribute which then points to the relative record concerned. The relative record number is then converted to track and sector addresses on disk.

Random Access Programming of a disk is always a question of producing by whatever means possible, a physical disk address (track, sector and buffer positions in the case of 'packed' records) in response to a request for data.

Often sequential files are kept to either point directly to a record (or even another index) or to contain information about a record (live or deleted) as in this last case for CBIS, CSTOCK, etc. and as does the BAM of system level in the 2/3040 disk.

The most widely used technique is the Index Sequential Method - Martin here is superb. The other principal method of file structuring is linked lists: here sectors contain pointers to the next sectors in the file. It is up to the programmer to decide what technique (or mixtures of techniques) is most desirable for a given application.

To come down from this to basics, if you simply want to get going with BLOCK-READ and BLOCK-WRITE, use the program below. Bear in mind that in any real world application, you will probably be using all sorts of exotic techniques to compute Track and Sector.

```
10 OPEN1,8,5,"#"
20 OPEN15,8,15
30 FORT=1TO3
40 FORS=0TO20
50 PRINT#15,"B-P";5;1
80 X$=STR$((T-1)*21+S)+"TESTING"
90 PRINTX$
100 PRINT#1,X$;CHR$(13);
110 PRINT#15,"B-W";5;1;T;S
120 NEXT
130 NEXT
140 CLOSE1
150 OPEN2,8,5,"#"
160 FORT=1TO3
170 FORS=0TO250
180 PRINT#15,"B-R";5;1;T;S
190 INPUT#2,A$:PRINTA$
200 NEXT
210 NEXT
220 CLOSE2:CLOSE15
230 END
READY.
```

DIRECT ACCESS

Mike Gross-Niklaus

1. AIM

While the Disk Operating System for the CBM 3040 disk drive supports sequential files to a greater extent than it does direct access files, all the tools for handling direct access are present. Provided you understand what they are and what they can do, they are quite simple to use and flexible enough to build up sophisticated direct access systems.

This article explains the various components of a direct access system, how to link them together and how to code them into your programs. The system described is simple, in fact it is the one used as an example on our disk utilisation course. Once you understand it, the way is open for you to design and implement your own made-to-measure direct access disk routines.

2. THE WAY INFORMATION IS ORGANISED ON THE DISKETTES

The information recorded on the diskette is organised into a series of tracks and, within each track, into a set of compartments called sectors. There are 35 tracks on each diskette. The number of sectors per track varies. The outside tracks, the longest, have 21 sectors per track and the inside ones, the shortest, only 17 sectors. The total number of sectors on the diskette is 690. An alternative name for a sector is a block. Conventionally one refers to information being contained in track 2, sector 3 for example or in block 25. Each block contains space for 255 characters.

A collection of items of information is called a file. Let's suppose you have a file containing the surname, first name and telephone number of all your business contacts and that these are held in alphabetical order. The details for each person in the file are collectively called a record. The pieces of information within each record are called items. In our example there are three items in each record.

3. SEQUENTIAL

There are two main ways you can record this information onto the disk and retrieve it. One is called sequential and the other direct access. Each method has its advantages and disadvantages. Part of the task of planning a disk based program or programs suite is to decide which method to use for holding particular files.

Let us consider sequential files first. In sequential recording, the Disk Operating System looks after the organisation of information on the disk into tracks and sectors. You tell the DOS what the next item of information to be recorded is and the DOS records it starting at the 'next available' character position on the disk, stradling two blocks if necessary, and avoiding blocks already in use by other files. Diagram 2 illustrates the concept.

One advantage of sequential organisations is that where your items vary in length, becuase they are packed one after the other on the diskette, no space is wasted.

A disadvantage of sequential files is that no record is kept to show you at what character in which sector of a particular track a specified item begins. Thus to read a particular item from a sequential file on the disk, it is necessary to read everything in the file which was recorded ahead of the required item, either counting items or looking for information which will uniquely identify it.

Using our example file, the program will have to search perhaps through many records before finding the details for ZAKS, RODNEY. Moreover, supposing you then want the phone number for SPENCER C, the programme will have to start over from the beginning of the file and make another long search. Also, the only way to amend an item in a sequential file is to write an updated version of the entire file.

Don't get the idea that sequential files are useless or second best, however. There are many situations where they are ideal. For example, when processing a payroll, where personal details and brought forward totals are matched against current information on hours worked and rates, both brought forward, current and carry forward files are ideally held in sequential format.

4. DIRECT ACCESS FILES

The other mode of operation is called direct access. It allows you to specify into which sector on which track and starting at which character a particular item of information is to go. And provided you keep a note of where you put the information, you can directly access it at a later time, without having to wade through all the other records. Because this allows you to examine items in the file in a non-sequential, unpredictable order, direct access files are often termed random access files. Since you can both read from and write to a particular place on the diskette, you can amend items in direct access files without rewriting the whole file.

Currently the disadvantage of direct access is that DOS doesn't help you a great deal. You must keep track of where you put the information. The direct access file is not listed in the diskette directory, and indeed no name for it is required by DOS. If you put your direct access file on a diskette containing programme or sequential files, you must take precautions to ensure you don't write into blocks in use by those files.

It is normal when using direct access to make the items fixed length. In our example, we might specify the surname as a key word of 20 characters, the first name as 10 characters and the telephone number as 20. In use, MIKE and ALFRED will both occupy 10 characters 'slots' in their respective records. As a result, direct access files use more space than sequential access files to hold the same information.

5. THE COMPONENTS OF A DIRECT ACCESS SYSTEM

When building a direct access system,you must decide how long each fixed length item is to be. In our example, the items are 10 and 20 characters long. The total number of characters per record is 30, so it would be possible to fit eight records into each block. However, to keep things simple while you learn the techniques, the example will use 1 record per block only.

The components required to write one record per block are:-

a. At the start of direct access operations, open up a route from PET.
   to a buffer in the disk unit.
b. Copy the record from one or more BASIC variables into that buffer, starting at character position 1.
c. Find the next available block on the diskette.
d. Tell the DOS that the block is reserved.
e. Write the whole buffer to that block.
f. Make an entry in an index array relating the block to the record key. The record key is the index word you use to look up the record details. In our example it will be the surnames of the various people in the telephone file.
g. At some stage thereafter, save the index array as a sequential file.

Once the index has been saved, the system can be closed down, switched off even, and later opened up again.

To read a record from the disk requires the following components:-

h. Read the index back from the sequential file into a BASIC array.
i. At the start of direct access operations open up a route from a disk buffer to the PET.
j. Search the index for the keyword of a required record and note the track and sector number associated with it.
k. Read the whole of the sector specified in the index from the diskette into the reserved DOS buffer.
l. Copy the information from the buffer into one or more BASIC variables.

Finally, to amend a direct access record:-

m. Read the whole block into a disk buffer as shown in h-k above.
n. Point at the part of the buffer to be overwrittten.
o. Copy the new information into the buffer from one or more BASIC variables over-writing the specified portion of the information in the buffer.
p. Write the contents of the buffer back to the block it came from.

6. WRITING A DIRECT ACCESS RECORD

Opening up the command route

The disk unit is normally set up as device number 8 on the IEEE 488 information bus. It has built-in software which comes into operation when the device is attached to the PET and both are switched on. Associated with this software are several information routes, or channels, within the disk unit. Of particular interest to us here are channels 2 to 14 which are routes for information to be written onto or read from the disk, and channel 15 which is the route for commands to the disk software and any messages from it.

It is good practice to open the command channel near the beginning of the program and to close it just before the program ends. The syntax is OPEN logical file number,8,15. The logical file number is used by BASIC as a key to the other details to save you having to type them in each time. It can be any number from 1 to 255 but is commonly made the same as the channel number. Thus OPEN15,8,15 will open a logical command file which can then be accessed by PRINT#15.

Opening up a data route

DOS allows 5 disk data routes plus the command route to be open at one time. A data route can either be opened at the beginning of the program in the same way as the command channel, or if many files are to be opened in the program, opened and then closed around each access or set of access calls.
However, closing the file causes a number of time-consuming (up to 2 seconds total) processes to take place within DOS. It is better to leave the route open if you can. You need to reserve a buffer to hold the information going to or comming from the diskette and this is specified in the OPEN command. The syntax is OPEN logical file,8,channel,"#" where the '#' reserves the next available buffer and associates it with the channel. Suppose we use channel 2 then the statement would be OPEN2,8,2,"#"

Copying the data from a BASIC variable

Once the route is opened, information can be copied from a variable to the buffer associated with the data channel using PRINT# logical file number, variable name. For example, supposing the first name and phone number had been padded to 10 and 20 characters respectively and then concatenated into a 30 character string. (The keyword need not be part of the direct access record). e.g.

```
MIKE.....01 388 5702.......
```

If the formatted information were in A$, then it could be transferred to the buffer by PRINT#2,A$ assuming a data route had been opened as logical file 2.

In order to allow specific parts of the buffer to be overwritten, (see 5 m-p above), the information is written into the buffer starting at the character pointed at by a buffer pointer held within DOS. To alter the position of this pointer, we must command DOS using channel 15. All the DOS commands can be shortened to their initial letters. The syntax for moving the buffer pointer is, (where logical file 15 has been opened as the command channel):-

```
PRINT#15,"B-P";C;P
```

where C is the data channel number and P is the pointer position required. So a sequence for getting our telephone record into the buffer would be:-

```
PRINT#15,"B-P";2;1:REM BLOCK POINTER
COMMAND TO DOS

PRINT#2,A$          :REM DATA
```

Finding a free block

A simple way of finding spare blocks is to put the direct access files onto an otherwise empty diskette. Then, provided you don't use track 18, sectors 0 to 2, which are reserved for the disk directory, you control completely the placing of information on the diskette.

If you want to use a diskette which contains or will contain files placed there by DOS, then you will need to find out which blocks are free and reserve them. This is achieved in a rather indirect way, using the Block Allocate command.

When you use the Block Allocate command to tell DOS that a block is reserved, DOS will reply with a message which you can read from the command channel. If the block was free to be allocated, DOS will allocate it and reply with message Ø, "OK". If that block was already in use, DOS replies with message 65, "NO BLOCK", and the track and sector number of the next free block 'down the disk'. So you can always find the next free block by attempting to allocate track 1, sector Ø. The very first time that is done, it will be allocated, and from then on, the command channel will tell you where the next free block is. Having found it, we can allocate it, (which should give an OK message) and then write into it. From then on, when DOS writes its sequential or program files, it will avoid that block. The syntax to allocate a block requires the drive as well as the track and sector. Thus:-

```
PRINT#15,"B-A";Ø;1;Ø
```

attempts to allocate drive Ø, track 1, sector Ø.

```
INPUT#15,EN,EM$,ET,ES
```

reads the resultant message from the command channel. If EN = Ø then the block will be allocated. If EN = 65 then:-

```
PRINT#15,"B-A";Ø;ET;ES
```

will allocate the next free block which is track ET, sector ES.

Writing the contents of the buffer to disk

Again you use a DOS command, Block Write. The channel (and hence the associated buffer), the drive, the track and the sector must all be specified. The track and sector you already know. They are ET and ES. So:-

```
PRINT#15,"B-W";2;Ø;ET;ES
```

will write the contents of the buffer associated with channel 2 to track ET, sector ES on drive Ø.

Handling the index

The index associates the key-word with the track and sector number of the information you put on the disk. The best way of holding it is in array form. Since the track and sector numbers are numeric and the keyword is alpha, you can use two 'parallel' arrays, one string and one numeric or you could hold all three items in a three wide string array and convert the track and sector numbers from and to numeric form using STR$ and VAL. In our example, supposing there were just three entries, held in the three columns of I$(,), they would look like this:-

```
GROSS-NIKLAUS    1    Ø
SPENCER          1    1
ZAKS             1    2
```

If the next record added were that for GOLTZ, row 4 of the array might be:-

```
GOLTZ            1    3
```

Once the index gets large, searching from the beginning to find the required key word takes time, so at some stage, this array should be sorted into order, or you could arrange for each new item to be inserted into the correct position as it is entered. Once keys are in alphabetic order, it is possible to do a binary search to shorten the time to find one particular item. A binary search is similar to the 'Guess a number game' where one player (or the PET!) hides a number between one and a hundred, and you must guess it in as few turns as possible. In a binary search you would guess 5Ø to start with, then 25 or 75 depending on whether your guess was high or low. Similarly with an alphabetic binary search. You compare the keyword with the element halfway down the array. If the element is 'higher' than the keyword then you halve the gap and look at the element a quarter down the array, and so on until you find a match or until the next 'move' is less than 1, in which case the keyword is not in the index.

Reading data back again

First, get the keyword. In our example it might come from the keyboard using an INPUT statement. Then the index is searched, using a binary chop perhaps if the index is likely to be a long one, say over 2ØØ entries. Once the entry for the keyword is found, the track and sector number of the details are available.

The entire block is read into a buffer using "U1" in preference to "B-R" which tackles matters in a way inappropriate to this example. The same channel as that used for writing can be used. Thus:-

```
PRINT15,"U1";2;Ø;T;S
```

will read the information from track T, sector S of drive Ø into the buffer associated with channel 2.

7. THE 'NITTY GRITTIES'

INPUT#, like INPUT, can only 'bite off' chunks of data up to 8Ø characters long. It needs a 'delimiter' such as CHR$(13) or a comma to break up the data into such chunks and to signal the end of the last

chunk. PRINT#ing can put a delimiter on the end of a chunk for you, but if the chunk is more than 80 characters long or if you try to input more chunks than you wrote, BASIC gets indigestion so badly it will hang up! So ensure that when you write the data, it is suitably subdivided, and that you read back in the same 'pattern' as you wrote to the disk.

When you do print to the disk, using PRINT#2,A$ for instance, as well as a CHR$(13), a CHR$(10), (line feed) also gets recorded on the disk, and will head the second and subsequent chunks of data you read back. Assuming you don't want this, the technique is to print using the following syntax:-

```
PRINT#2,A$;CHR$(13);
```

which will put a CHR$(13) onto the disk but suppress the normal CHR$(13) & CHR$(10), because of the final semi-colon.

Avoid null strings, that is to say consecutive CHR$(13)'s, as part of your disk information. The best way is to test each string before you print it to the disk, using something along the following lines.

```
IFA$=""THENA$=CHR$(160):REM SHIFTED
SPACE

PRINT#2,A$;CHR$(13);
```

When a disk is VALIDATED, the Block Availability map, held on the diskette and used by the Block Allocate command to find the next free block, is cleaned up as part of the general tidy up that VALIDATE accomplishes. Since DOS didn't write the blocks in which you've recorded your direct access information, they are 'freed up' although the information in them remains intact. You can tackle the resultant problems by keeping your own availability map and reallocating all the blocks. Since the index itself is such a map, you can use this with a utility routine to re-establish your claim on those blocks before DOS goes trampling all over them. Some people prefer to keep their direct access files on a seperate diskette well away from Sequential and Program files.

8. TO SUM UP

Disk syntax is not as easy as BASIC, nor as forgiving. Mistakes in formatting information when writing to the disk often result in teeth grinding hang ups of BASIC as you read it back. But the steps are logical enough and the syntax is not hard to grasp.
If you are going to use the same diskette as DOS is using for your sequential and program files then you must be prepared to dodge out of the way of DOS as it goes about its busy and useful purposes, almost blind to the activities of us direct access mice about its feet.

PROGRAMMABLE LINE FEED

One of the very useful features of the 3022 Tractor Feed Printers, which has been relatively unrecognised so far, is the ability to control the amount by which the paper is advanced for each line feed.

This programmable line feed function has many uses, dot by dot graph plotting is one application which has already been mentioned in CPUCN. But perhaps most users will be interested in the printer's ability to create attractive form layouts for tables of information.

The table illustrating the principle of sorting by insertion and chaining in Mr Slow's article was printed on the 3022 and the program listing appears below.

Line 20 opens a channel to special secondary address 6 which is used to control the programmable line feed. Line 100 sets the line feed spacing so that the vertical lines just touch one another. Line 109 decreases the spacing still further before advancing to print the next pair of boxes.

Line 98 resets the line feed to the normal value, which is preset when the printer is first turned on.

```
10 OPEN4,4
20 OPEN6,4,6
40 GOSUB100
50 R$=" ":S$="1":T$="0":X$=R$:Y$=R$:Z$
   =R$:GOSUB200
60 R$=" ":S$="2":T$="0":X$=R$:Y$="1":Z$
   =R$:GOSUB200
70 R$="3":S$="2":T$="0":X$=" ":Y$="1":Z
   $=X$:GOSUB200
80 R$="3":S$="2":T$="0":X$="4":Y$="1":Z
   $=" ":GOSUB200
90 R$="3":S$="2":T$="0":X$="4":Y$="5":Z
   $="1":GOSUB200
98 PRINT#6,CHR$(24)
99 END
100 PRINT#6,CHR$(18)
103 PRINT#4,"    | 0 | 1 | 2 | 3 | 4 |
    5 |...|32 |...|50 |"
104 PRINT#4,"----+---+---+---+---+---+
    ---+---+---+---+"
105 PRINT#4,"B%  |   |   |   |   |   |
      |   |   |   | 0 |"
106 PRINT#4,"----+---+---+---+---+---+
    ---+---+---+---+"
107 PRINT#4,"C%  |   |   |   |   |   |
      |   |   |   |"
108 PRINT#4,"----+---+---+---+---+---+
    ---+---+---+---+"
109 PRINT#6,CHR$(10):PRINT#4
110 RETURN
200 PRINT#6,CHR$(18)
204 PRINT#4,"----+---+---+---+---+---+
    ---+---+---+---+"
205 PRINT#4,"B%  | "R$" |   |   |   |
      |   |   | "S$" |   | "T$" |"
206 PRINT#4,"----+---+---+---+---+---+
    ---+---+---+---+"
207 PRINT#4,"C%  |   |   | "X$" | "Y$" |
    |   | "Z$" |   |   |   |   |"
208 PRINT#4,"----+---+---+---+---+---+
    ---+---+---+---+"
209 PRINT#6,CHR$(10):PRINT#4:PRINT#6,
    CHR$(18)
210 RETURN
```

UPPER/LOWER CASE CONVERTER

Paul Higginbottom

This program converts upper case to lower case and vice versa. So it will save a lot of trouble when converting programs to run on New/Old ROMs. Input the program and remember to save it before running. When the program is debugged it can be used at any time to change the case of another program by being loaded and run, this will load the machine code into the second cassette buffer. The program to have its case changed is then loaded and SYS826 is typed. The program can then be resaved. Simple!

```
10 PRINT"■    ■UPPER/LOWER CASE CONVERT
   ER"
20 PRINT
30 PRINT"■■THIS MACHINE CODE PROGRAM CON
   VERTS"
40 PRINT"■■THE SCREEN TEXT OF A PROGRAM
   WRITTEN"
50 PRINT"■■FOR THE OLD PET INTO THE NEW
   VERSION"
60 PRINT"■■AND VICE VERSA.  ALL YOU HAVE
    TO DO IS"
70 PRINT"■■LOAD THIS PROGRAM AND RUN IT.
    THEN"
80 PRINT"■■LOAD THE PROGRAM TO BE CONVER
   TED AND"
90 PRINT"■■TYPE 'SYS826'.  THEN RESAVE T
   HE PROGRAM."
110 FOR J=826 TO 906: READ A: POKEJ,A:
    NEXT
130 DATA169,4,133,202,169,1,133,201
140 DATA32,89,3,160,0,196,202,240,13
150 DATA177,201,170,200,177,201,134
160 DATA201,133,202,76,66,3,96,160,4
170 DATA177,201,240,249,201,34,240,4
```

```
180 DATA200,76,91,3,200,177,201,240
190 DATA236,201,34,240,23,201,65,144
200 DATA243,201,91,144,4,201,192,144
210 DATA235,201,219,176,231,73,128
220 DATA145,201,76,103,3,200,76,91,3
230 NEW
```

PRINT USING

Dave Middleton

Anybody who has used the PET to output numeric data in tables will know how frustrating it can be to get the columns to align. A table with the followng format is all too common:

```
 No    Position      Time
0      1210153312    153.11123
10    12.312          16.215
17     .1273          5.12563127
5       150             0
```

There are quite a few one line routines which will chop a number around so that it will fit a set format, but none that I have seen will allow true table layout.

I required a routine which would allow me to output a large amount of data into a table that would just fit an 80 column printer. The following program is the result. It works with disk files but it can easily be modified by changing the channel number so that it reads from tape. i.e. change 8 to 1 in line 270 and miss off the rest of the line:-

```
270 OPEN5,1
```

Lines 140-630 have to be altered to give the required data. The input routine shown is for an engine test bed running on petrol or methanol.

Lines 150-260 are the program constants some of which are sent to the printer.

Lines 370-450 perform calculations on the input data.

Lines 460-600 send the data to disk.

When modifying the program it is only necessary to ensure the following items:

1. The first item to be sent to disk for each data block is 1. Line 460 sends CH to disk: CH=1

2. The first item to be sent before closing the disk file is 99999 as in line 630.

3. The title for the output data is held in F$.

4. The data format is held in P$ using 'I' to denote Integer output and 'F' to denote Floating point. Spaces act as delimiters between numbers.

5. Ensure that the number of items sent to disk for each line matched the number of format fields in P$!

Any number out of range is shown by '##' in the printout.

HOW THE PROGRAME WORKS

Take a simple output format string to be:

```
P$= SIISSSFF.FFFSSSI"
```

Each set of data to be output will be stored on disk as a group of 3 preceded by a check digit and the data block will be terminated by 99999. Thus for example there are three sets of data to be output this will stored as follows:

```
1 5 2.3716215 3 1 10.1 2021 9 1 69
95.6275 4 99999
```

740
The first item is read from disk. If it equals 99999 then there is no more data and the program terminates. The number is discarded after the test.

The position pointer CO is set to the start of P$.

750
The character under the pointer is assigned to V$. The pointers FB and FA (Float Before, Float After decimal point) are set to 0. In the above example V$=" ":CO=1.

770>
A space is printed and the program jumps to 990 which will increment CO.

This is repeated until CO-3 V$='I'

790
The string is now searched until a space is encountered or the end of the string is found.

810
FB is set to the number of I's encountered.

1100
A number is read from disk.

1030/1040
The number is turned into an integer NI and NI is turned into a string. If the length of N$ is greater than the number of characters set aside for it in FB or if N>IEIO an error message is printed and # symbols are sent to the printer.

1060
When a number is converted into a string it will have a preceding space if it is positive and a '-' if negative.

I have written the routine so that this space is removed for positive numbers thus saving space which is important on tightly packed output. Thus the user has to include an extra I or F into P$ when negative numbers are going to be output.

1070
If the number is negative then the - sign is put into the first position of the format. eg. N=-50

```
IIIII
- 50
```

1080
If the number is positive then blanks are added. Thus numbers are always right justified.

The sample printout will now look like:

```
SSS
```

770
More spaces are encountered and sent to the printer. When the first 'F' is found CO=8.

840
The string is searched. FB is incremented when FA=0.

850
As soon as the decimal point is found then FA is incremented.

860
When FA>0, FA is incremented.

870
When the next space is found the program branches.

900
A Floating point number can be considered as an integer + a bit left over. Hence the number before the decimal point is sent to the printer using the integer subroutines in 1030. A decimal point is printed. CS is set to 0.

910
As soon as a number drops below .01 it is output in exponential format. When this happens it is necessary to convert it back into floating format by adding in some zeros.

920
CS is set equal to the exponent. eg

```
1.237 E- 05
  N      CS
```

The actual number is made to conform to the floating point remainder by dividing IO ie:

```
N=0.1237
```

950
If 0's are now added between the point and the next number it will be made to conform to true Floating point format. ie:

```
0.00001237
```

970
If CS > FA then only 0's will be output. Starting from the decimal point in N numbers are printed. If the numeric output has still not filled the FA field then trailing '0"'s will be printed to pad it out.

980
CO is incremented by the length of the format FB+FA

There are two operations that the program will not perform, these being alphanumeric and exponential output which brings me to the competition.

Modify the program so that alphanumeric data can be input and formatted and add a routine which will convert any number into exponential format. Use A in P$ for alpha and E for exponential. I would suggest that a block of E's be used with a minimum of 8 being necssary to give output. ie:

```
EEEEEEEEEE
-1.763E-10
```

Set up the program so that it uses cassette Input/Output and will format to the screen (OPEN1,3) send your modified program to me, Dave Middleton, enclosing a SAE for return of your cassette. The closing date is 1.10.80 and the prize is any program from the Master Library.

```
10 REM******NOTE - ALL REM STATEMENTS
11 REM**********MAY BE OMMITED**********
20 REM**************START***************
95 REM*********F$ IS TITLE STRING******
96 REM**********P$ IS FORMAT STRING******
100 F$="RUN SP   N  LOAD    PH    MA
    MF   AFR  Y.VOL   TQ    BP    VB  SF
    C"
110 F$=F$+"    BMEP"
120 P$=" II II  F.F FF.FF FF.FF FF.FF FF
    .FF FF.FF FFF.F FF.FF FF.FF FF.FF F.
    FF"
130 P$=P$+"FF FFF.FF"
132 REM
135 REM******INPUT MAIN VARIABLES******
140 OPEN10,4
150 INPUT"DATE";DA$:PRINT#10,"DATE
      ";DA$
160 INPUT"JET SIZE";JE$:PRINT#10,"JET SI
    ZE    ";JE$
170 INPUT"NEEDLE TYPE";NT$:PRINT#10,"NEE
    DLE TYPE ";NT$
180 IFTY$="M"THENPRINT#10,"FUEL TYPE: ME
    THANOL"
190 IFTY$="P"THENPRINT#10,"FUEL TYPE: PE
    TROL"
200 CH=1:INPUT"ATOMSPHERIC PRESSURE";PA
210 PRINT#10,"ATMOSPHERIC PRESSURE";PA
220 INPUT"FUEL TYPE ■M■ETHENOL OR ■P■ETR
    OL";TY$:IFTY$="M"THENCV=30000:FD=700

230 PRINT#10,"FUEL TYPE: ";:IFTY$="P"
    THENPRINT#10,"PETROL"
240 IFTY$="P"THENCV=43900:FD=753
250 IFTY$="M"THENPRINT#10,"METHENOL"
260 PRINT"CALORIFIC VALUE";CV:PRINT"FUEL
     DENSITY    ";FD :CLOSE10
264 REM
265 REM**********INPUT DATA**********
270 OPEN5,8,9,"@1:STEF,S,W"
280 INPUT"RUN NUMBER      ";C0
```

```
290 INPUT"PISTON HEIGHT MM";PH
300 INPUT"SPEED  1000RPM  ";N
310 INPUT"LOAD       LBS  ";L
320 INPUT"DELTA H  MMH20  ";H
330 INPUT"TEMP        °C  ";T
340 INPUT"VOLUME      ML  ";V
350 INPUT"FUEL TIME  SEC  ";TS
360 INPUT"SPARK TIME °BTC ";SP
364 REM
365 REM******PERFORM CALCULATIONS******
370 MA=4*SQR((H*PA)/(T+273))
380 MF=(FD*V)/TS/1000
390 AF=MA/MF
400 YV=51.78*(SQR((H*(T+273))/PA))/N
410 TQ=1.446*L
420 BP=0.1514*L*N
430 YB=(BP*100000)/(MF*CV)
440 SF=8.2/YB
450 BM=9.116*L
454 REM
455 REM*******PRINT DATA TO DISK*******
460 PRINT#5,CH:PRINT#5,C0:PRINT#5,SP:
    PRINT#5,N:PRINT#5,L:PRINT#5,PH:
    PRINT#5,MA
470 PRINT#5,MF:PRINT#5,AF:PRINT#5,YV:
    PRINT#5,TQ:PRINT#5,BP:PRINT#5,YB
590 PRINT#5,SF:PRINT#5,BM
595 REM
596 REM********MORE DATA ?**************
610 INPUT"ANY MORE DATA■■■■■■■■";AN$
620 IFAN$<>"N"GOTO280
630 PRINT#5,99999:CLOSE5
644 REM
645 REM******START OF PRINT USING******
650 OPEN10,4:OPEN4,8,3,"1:STEF,S,R"
654 REM
655 REM*********PRINT TITLES*************
660 FORA=1TO80:PRINT#10,"*";:NEXT:
    PRINT#10
670 PRINT#10,F$
680 FORA=1TO80:PRINT#10,"*";:NEXT:
    PRINT#10
693 REM
694 REM*******MAIN PROG STARTS HERE***
695 REM******TEST FOR END IE 99999*****
740 CO=1:GOSUB1100:IFN=99999GOTO1010
744 REM
745 REM***START OF NEXT FORMAT FIELD**
746 REM*****TREAT SPACE AS AN END******
750 V$=MID$(P$,CO,1):FB=0:FA=0
760 IFV$=""GOTO990
770 IFV$=" "THENPRINT#10," ";:GOTO990
780 IFV$<>"I"GOTO830
784 REM
785 REM*********INTEGER ROUTINE********
790 FORC1=COTOLEN(P$):IFMID$(P$,C1,1)="
    "GOTO810
800 NEXTC1:FB=LEN(P$)-CO+1:GOTO820
810 FB=C1-CO
820 GOSUB1100:GOSUB1030:CO=CO+FB-1:GOTO9
    90
823 REM
825 REM*****FLOATING POINT ROUTINE*****
830 IFV$<>"F"GOTO63999
840 FORC1=COTOLEN(P$):D$=MID$(P$,C1,1):
    IFD$="F"ANDFA=0THENFB=FB+1
850 IFD$="."THENFA=1
860 IFD$="F"ANDFA>0THENFA=FA+1
870 IFD$=" "GOTO900
880 NEXTC1:IFFA>0THENFA=FA-1
900 GOSUB1100:GOSUB1030:PRINT#10,".";:CS
    =0:S=N
910 IFABS(N)>1E-2GOTO940
920 CS=INT(VAL(RIGHT$(STR$(N),2))):N=
    VAL(LEFT$(STR$(N),LEN(STR$(N))-2))/1
    0
930 IFCS>0THENCS=CS-1
940 IFCS=>FATHENCS=FA-1
950 PRINT#10,LEFT$("000000000",CS);
960 V=2:IFABS(N)<1THENV=1
970 PRINT#10,MID$(STR$(N)+"00000000",
    LEN(STR$(INT(N)))+V,FA-CS-1);
980 CO=CO+FB+FA-1
985 REM
986 REM*****FINISH FORMAT ON THIS*****
987 REM******NUMBER. INCREMENT CO******
990 CO=CO+1:IFCO=>LEN(P$)THENPRINT#10:
    GOTO740
1000 GOTO750
1005 FORA=1TO80:PRINT#10,"*";:NEXT:
     PRINT#10
1010 PRINT#10:CLOSE10:CLOSE2
1020 STOP
1027 REM
1028 REM***TURN NUMBER INTO AN INTEGER
1029 REM*PRINT ERROR IF GREATER THAN FB
1030 NI=INT(N):N$=STR$(NI):IFLEN(N$)<FB
     +2ANDNI<1E9GOTO1060
1040 PRINT"ERROR NUMBER OUT OF RANGE":N:
     N$=LEFT$("##########",FB):GOTO1090
1060 N$=RIGHT$(N$,LEN(N$)-1)
1070 IFSGN(N)=-1THENN$=LEFT$("-
     FB-LEN(N$))+N$:GOTO1090
1080 N$=LEFT$("          ",FB-LEN(N$))+N$
1090 PRINT#10,N$;:RETURN
1097 REM
1098 REM****READ A NUMERIC STRING FROM
1099 REMDISK CHOP OF CHR$(10) AT START*
1100 INPUT#4,CH$:CH$=MID$(CH$,2,200):N=
     VAL(CH$):RETURN
```

BITS AND PIECES

PRINTER TABBING

When using TAB to print on the screen, PET looks at the current position of the cursor first (POS(Ø)). If the TAB

```
DATE          30 JULY 1980
JET SIZE      1.001
NEEDLE TYPE 1510
ATMOSPHERIC PRESSURE 761
FUEL TYPE: METHENOL
*************************************************************************************
RUN SP   N  LOAD     PH     MA     MF    AFR  Y.VOL    TQ     BP     YB  SFC     BMEP
*************************************************************************************
  1  0  1.0 25.00 25.00 51.10 70.00  0.73 263.4 36.15  3.78  0.18 #.4953 227.9
  2  0  2.0 29.00 29.00 36.83 22.55  1.63 101.2 41.93  8.78  1.29 6.3188 264.3
  3  5  3.0 60.00 50.00 41.82 14.00  2.98  82.5 86.76 27.25  6.48 1.2637 546.9

*************************************************************************************
```

argument is less than the cursors' position on the line then the data is simply printed in the spaces immediately following the last character printed. If the argument is greater than or equal to POS(Ø), PET subtracts POS(Ø) from the argument and prints the resulting number of cursor-rights.

However, when printing to the printer, the cursor is usually in column zero and TAB acts like the SPC function (the printer has no "internal cursor"). Therefore, to make TAB work on the printer, print the data to the screen first then to the printer. This can be done with duplicate PRINT and PRINT# statements or more efficiently with one "dynamic" PRINT# statement. For example:

```
1Ø REM OPEN OUTPUT FILES TO
   SCREEN & PRINTER
2Ø OPEN 3,3,1
3Ø OPEN 4,4,Ø
4Ø PRINT# 3+X,"ABCDEFGHIJKLMNOPQRSTUV
   W";5Ø X=1-X : IF X THEN 4Ø
```

Line 5Ø toggles X from 1 to Ø thus repeating line 4Ø only twice. The semi-colon is important else the POS(Ø) goes back to zero. When a carriage return is required on the printer the following might be inserted between PRINT# and toggle statements:

```
45 IF X THEN PRINT#4,CHR$(13);
```

Dynamic PRINT# statements are only more efficient if the DATA being printed is within quotes. If variables are used, more bytes are probably saved by duplicating the output statements.

MORE ON PRINTER OUTPUT

L. D. Gardner of Leisure Books Ltd., St. Laurent Quebec, wrote in with:

Dear Sirs,

Our company uses two 32k Commodore Pets with Centronics printers. *

The printers are used to produce formal reports of a statistical nature, and we have found that the most difficult task is not in programming the calculations, but in printing the data in acceptable format.

If 2 decimal places are required, a value of "3Ø2" is not printed as "3Ø2.ØØ", which makes the reports difficult to read. Also, a very small value such as ".ØØ1" is printed as "1E -Ø3".

I have come up with a subroutine to handle this problem. Undoubtedly, it could be shortened, but for clarity is is shown in the detailed version.

```
10 REM ********************************
20 REM ***SUBROTUINE FOR TO FORMAT ***
30 REM ***NUMBERS TO 3 SIG. DIGITS ***
40 REM ********************************
100 Q$=".000":LQ=LEN(Q$):LT=LOG(10)
110 P=LQ-1:IF Q$="" THEN P=0
150 INPUT"NUMBER",A
160 A1=INT(A*10↑P+.5)/10↑P
165 IF A1=0 THEN220
170 B1$=STR$(A1)
180 B2$=STR$(A1*10↑P)
190 LG=INT(LOG(ABS(A1))/LT)
195 IF ABS(A1)=1 THEN B$="1"+Q$
205 IF ABS(A1)<1 THEN B$=LEFT$(Q$,ABS(L
    G))+RIGHT$(B2$,LEN(B2$)-1)
215 IF ABS(A1)>1 THEN B$=MID$(B1$,2,LG
    +1)+"."+RIGHT$(B2$,P)
216 IF ABS(A)=1 THEN B$="1"+Q$
220 IF A1=0 THENB$=Q$
230 IF A1<0 THENB$="-"+B$
240 PRINT"@@",A1,B$
250 GOTO150
```

SUBROUTINE TO FORMAT OUTPUT DATA

* This would also apply to Commodore printers and others

LIST IN LOWER CASE

Robert Oei, a PET user in Mississauga, has discovered a sequence that will cause PET printers to LIST in upper/lower case rather than graphics and upper case...

The method to accomplish this is actually quite simple:

1. POKE 59468, 14
2. Enter the following line in immediate mode:

```
OPEN1,4,1:OPEN2,4,2:PRINT#2,"9":FORA=1TO
2:PRINT#1,A;A;A;A;A;A:NEXT:CLOSE1:CLOSE2
```

After pressing RETURN, the printer will print six "1"s and then six "2"s on the next line. If a file is then opened to the printer and the "cmd" command is given, the printer will respond with the word "ready." in lower case. Any program listing performed from then on will appear in upper/lower case. Power down and up to get graphics back.

I regret to say that I don't know what caused this phenomena to occur, except that it works. I would be most interested if you or any PET user can provide a logical reason as to why?... Robert Oei

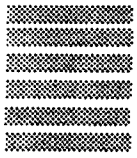
This was tested on 2Ø22/23 printers and found to work on some but not all.

PRINTER FIX

E. Smart

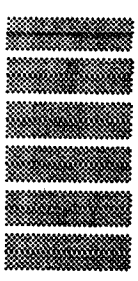
There is a small bug in the printer operating system in that it will not respond to a second command sent immediatly to the same secondary address. The following program illustrates the problem:-

```
 100 OPEN4,4: OPEN6,4,6
 110 FOR I=1 TO 3
 120 PRINT#6,CHR$(24)
 130 PRINT#6,CHR$(16)
 140 PRINT#4,"▒▒▒▒▒▒▒▒▒▒"
 150 PRINT#4,"▒▒▒▒▒▒▒▒▒▒"
 160 NEXTI
 170 CLOSE6
 180 CLOSE4
READY.
```



Notice all the lines are the same distance a apart. Line 130 is telling the printer to close the gap between the lines up but it is being ignored. This can be corrected by adding a PRINT to a secondary address of 0. This is achieved by line 125 which sends a carriage return without a line feed, the semi-colon stops the 'end of print' line feed being sent.

```
 100 OPEN4,4: OPEN6,4,6
 110 FOR I=1 TO 3
 120 PRINT#6,CHR$(24)
 125 PRINT#4,CHR$(141);
 130 PRINT#6,CHR$(16)
 140 PRINT#4,"▒▒▒▒▒▒▒▒▒▒"
 150 PRINT#4,"▒▒▒▒▒▒▒▒▒▒"
 160 NEXTI
 163 PRINT#6,CHR$(24)
 168 PRINT#4,CHR$(141);
 170 CLOSE6
 180 CLOSE4
READY.
```



Lines 163 and 168 are required to set the printer back to standard line feed of 6 lines/inch, RUN does not cause the line feed to be reset to standard.

QUIETER WRITER

There are some wing nuts in the bottom of the Tractor printer. These are only there to secure the Printer mechanism in transport and therefore are not required for general use of the Printer.

Their removal will make the operation of the Printer quieter. We recommend you keep the wing nuts in a safe place in case you wish to transport your Printer around at any time.

Jim Butterfield and Brad Templeton discovered a method by which two programmes could be merged together. With one stored in RAM and the other stored on cassette as a CMD listing. The method for achieving this was described in an early edition of CPUCN. The following method is both simpler and bug free.

MERGING PET PROGRAMS

Jim Butterfield

To wrap up the various activities surrounding merging or UNLIST and bring them up to date with information for the new ROM.

1. To change a program into a data file on casette tape. Mount a blank tape on casstte 1 and type:

```
OPEN1,1,1: CMD1: LIST
```

The cassette tape will write an ASCII listing of the program onto the tape. When the writing has finished, the flashing cursor will return but READY will not be printed, in fact READY has been written onto the tape! Now close the CMD file with:-

```
PRINT#1: CLOSE1
```

This "merge" tape may now be saved for future occasion.

Variations:
----the file may be named, eg.
OPEN1,1,1,"TEST MERGE"
----if desired only part of the program may be saved on tape.
CMD1: LIST500-700

2. To merge a data file (in the above format) back into a program. The procedure is slightly different on the Old ROM as compared to the New ROM. The program with which you wish to merge must first be loaded into memory. The following procedure may then be repeated many times so you may merge several program blocks together.

Mount the merge tape on cassette 1 and type:
Old ROM: POKE3,1: OPEN1
New ROM: POKE14,1: OPEN1
The tape header will now be read and eventually the computer will report FOUND and the cursor will return.

Now clear the screen and press three cursor downs only then type:-
Old ROM:
POKE611,1: POKE525,1: POKE527,13: ?"h"
New ROM:
POKE175,1: POKE158,1: POKE623,13: ?"h"

("h" is the cursor home key - it will print as reverse S)

As soon as you press return at the end of the line, the word READY will appear above the line and the cassette tape will move. When the merge is complete the computer will print out ?OUT OF DATA ERROR or ?SYNTAX ERROR below the line. This is normal and does not signify a real error. The job is not complete.

The new system is simpler and also corrects a minor problem on the original merge. Few people spotted it but the original procedure caused line 1 to disappear.

EXTRA USE OF VERIFY

VERIFY has the useful property of reading a program without loading it. Even if an error is detected it is not displayed until the whole program has been read.

This provides a quick means of finding the end of a program on a cassette without losing what you currently have in memory.

For example, one is occasionally in the position of having loaded a program, rewound the tape, modifed the program and now wish to save the new version after the old. Using an extra tape this can be done using LOAD and SAVE, with a bit of shuffling.

It is far quicker to use the VERIFY command. When the message appears you know that you are at the end of the last program. Press STOP and then SAVE the new version using a different name to that of the old version.

DATA FILE ERRORS (OLD ROMS)

There is a bug in the file handling routine which causes data to be written on the tape prematurely, not allowing for cassette motor start up time.

This is temporarily curable by keeping the motor running whilst the tape buffer is being filled, or by starting the motor when the buffer is almost filled.

The method of turning on the motor is to change a bit in the appropriate PIA register. The location of the PIA register is 59411 and the correct byte to place in that register is 53. Therefore the syntax for turning on the cassette motor is POKE 59411,53. This should be done either every time PRINT# is used or just before the buffer is full. Using the latter method involves PEEKING location 625 which is the buffer pointer. When this pointer approaches 191 which is the size of the buffer, turn on the motor. The relevant locations of bytes for the second cassette port are 59456 and 223 for STOP and 207 for START.

A problem with opening files to write on either built-in cassette #1, or external cassette #2, has been discovered. When a file is opened, garbage will be written out instead of a proper data tape file header. Without this header, it is impossible to open the tape file for reading.

You may not have encountered this problem previously, because it is disguised by having loaded a program on the cassette prior to writing a data file. In this mode, the start address of the buffer with the header information is initialised properly but cassette data file operation still could be random.

Fortunately, there is a software patch you can implement in your BASIC program to force the open for write on tape to work every time.

Before opening to write on #1 cassette:

```
POKE 243,122
POKE 244,2
```

And on #2 cassette:

```
POKE 243,58
POKE 244,3
```

Locations 243 and 244 contain the lo and hi order bytes respectively of the address of the currently active cassette buffer. The start address of buffer #2 is $33A which is 3,58 ($3=3,$A=58) in double byte decimal. Similarly, cassette #1 is $027A ($2=2, $7A=122).

TAPE HEAD CARE

It has been noted that the READ/WRITE head in the PET cassette deck has the annoying habit of magnetising itself after a remarkably short period of operation. It is in fact possible to partially erase your tapes by up to 15% after only 10 or 20 passes over the head. The most convenient way to deal with this problem is to demagnetise the tape head very frequently, i.e. every couple of days with a demagnetising cassette. AMPEX market quite a good one for about 3.00 pounds.

DUPLICATING CASSETTES FOR COMMODORE

One or two of you have expressed concern in the past about the quality of the pre-recorded cassettes that we send out from Commodore. Well, your worries are over. Cassette duplicating is now being done for Commodore by a company called Audiogenic from Berkshire. The quality of the first batch received from them has been superb, and as from January 1st, all tapes from us will have been recorded by Audiogenic.

Their director, Martin Maynard, takes up the story.

Before going into detail, may I introduce myself as Martin Maynard of Audiogenic Limited situated in Reading. My company was called in to duplicate and package the cassette based software produced by Commodore. Although Audiogenic has duplicated music cassettes for the past five years, digital tapes are something new to us, and fortunately, I was able to call upon my past experience in the data communications industry. The cassette deck containing moving parts, is probably the weakest part of any microcomputer system, and likely to be the first area upon which suspicion will fall when difficulties in loading a program are experienced. With this in mind, it will be of interest to PET users to know how the signal is recorded on tape and to what length Audiogenic goes to, to ensure the entire range of PET pack Software will load first time.

The PET cassette deck uses an unequalised recording method to place data on tape, by switching the direction of current through the record head saturating the tape either negatively or positively. The encoding scheme uses three distinct full cycle pulses (see fig. 1), a data zero or one is represented by a pairing of a short and a long pulse. If the shorter pulse is first, the pair is considered a one. The byte market provides reference for byte indentification. (See fig. 2). In the playback circuit the recorded signal passes through equalisation and squaring circuits, thus logic level signals are presented to the PET. The PET measures between negative going edges of signals and decodes the data from these measurements.

When we are duplicating tapes, we have to be very conscious of the slew rate of our negative going pulses, as it will be seen that if the slew rate is slow, there is a larger area of indecision presented to the squaring circuits whose threshold is set about the zero crossing point of magnetic flux on tape. After tapes have been duplicated, they are subject to quality control procedure, every one in six tapes is checked using Commodore's "Tape Graph 21S". This programme is loaded in the normal method and when running, examines a signal being received

on cassette port one, and measures the timing between negative going pulses and displays the results on the screen as a bar graph (see fig. 3.), showing up any nasties that may be occurring, with the timing due to uneven tape speed, dirt in the capstan, dirty head and spurious noises, all of which affect the timing on a tape.

Out of each batch of sixty tapes one tape has all its programmes loaded, and the "PRINT PEEK (630) ST" test is used. (See owner's manual), this must give a 0-0 reply. If all the samples pass this test that batch of tapes is visually inspected and packaged. If any failures are detected, the entire batch is rejected. Listening to a PET tape on your hi-fi set is also revealing, lack of high frequency content will indicate dirty or magnetised heads, and speed variations can be detected as overall pitch drifting.

It should be stressed that whilst we take great care in producing "Work first time Software", the system is only as good as its weakest link, which is the cassette deck, and the user should clean and demagnetise both the heads on the deck at least every five hours of operation. Heads are best cleaned with cotton buds and alcohol, and the capstan pinch roller should be cleaned. Do not rely on TDK cassette demagnetisers as they do not degauss the erase head. When making your own tapes on a PET always use quality audio tapes as budget tapes suffer from drop out and mechanisms fail to run smoothly. Any of the leading makes of low noise tapes are acceptable of course, official Commodore blank tapes can be purchased from our dealers.

by Martin Maynard
Audiogenic Limited
34/36 Crown Street
Reading
Berks

Figure 1

Data Timing

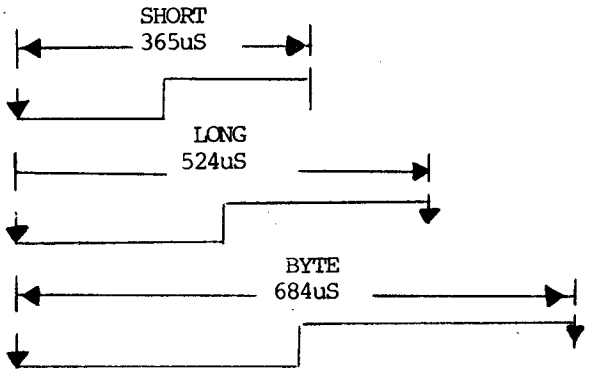


Figure 2

Data In/Out

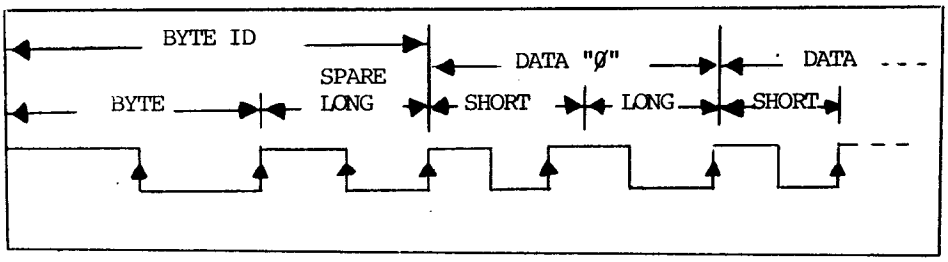
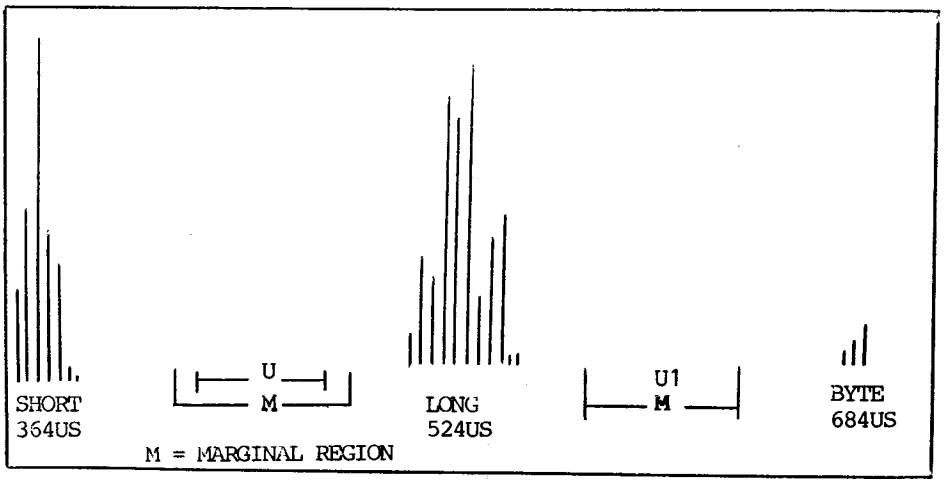


Figure 3



WATCHING A CASSETTE LOAD

Jim Butterfield, Toronto

It may not be too useful, but it's very satisfying to watch a program coming in from cassette tape. Much of what comes in will look like gibberish, since the program contains obscure things like pointers, flags and tokens. But it's interesting to see, and here's how you can do it.

Step 1: Load any Basic program on cassette 1. The program doesn't matter; the LOAD activity sets up certain internal things that will help us.

Step 2: Set up the cassette with any BASIC program ready to load. A short one would be good: that way you may catch the whole program on the screen. But any BASIC program will do.

Step 3: Set graphic mode with POKE 59468,14. This may help you spot a few recognisable pieces of your program.

Step 4: Give SYS 62894. PET will ask you to press PLAY. Do so, and in twenty seconds or so, PET will report FOUND ..... and stop.

Step 5: Clear the screen so you'll get a better view of the program as it comes in. Now move the cursor down to a few lines from the bottom of the screen.

Step 6: Enter POKE 636,128:POKE 638,132: SYS 62403. (Old ROM 62278).

Step 7: Sit back and watch the program load to the screen. You won't be able to run it, of course, since it's in the wrong part of memory... but isn't it fascinating to watch?

THE 'UNWEDGE'—A TAPE APPEND AND RENUMBER PROGRAM

Many PET users have only tape input for their system. (Until a few weeks ago, I was among that majority). If you have often wanted to append a favourite subroutine, you may have been stymied. I use the input subroutine from Cursor #4 ('INP', Oct 1978) in many programs. You may have your own favourites, if only there was an easy way to append them.

Another possibility might entail merging several short programs together, such as the Osborne & Associates: 'Some Common Basic Programs'. Individually, most are quite short, which could permit a 'menu-driven' composite program.

I am sure that other applications may come to mind.

Most of us have a Renumber program of some sort, whether it is in Machine Language, or as a Basic subroutine which is tacked on to the end. The main disadvantage is that the location in memory is fixed.

Welcome to the Upgrade-ROM combination of the two, which offers:

***Machine Language for speed of operation.
***Full relocatability to anywhere in RAM. When run, 'APPEND/RENUM' places itself in high memory and protects itself from intrusion by Basic.
***The 'Unwedge' (or '<' key) attaches itself to the operating system. It may be used in the Direct Mode with a minimum of user input.
***It is compatible with the Disk Wedge routine. Although the Append function works only with tape input, both Wedges may be active simultaneously.
***The routine may be implemented and de-activated with the same 'SYS' command.
***Uses only 771 bytes of memory.

Bill Seiler deserves the credit for nearly all the coding. In Pet User Notes (Vol I,#7 Nov/Dec 1978), he presented his 'PET Renumber 3.0' for Original ROM. This has been converted to Upgrade ROM and allows for user input of parameters.

In our own Transactor (Vol 2,#3 July 1979), Bill presented the Append Wedge for Original ROM. In the same Pet User Notes issue, Jim Russo and Henry Chow gave us M7171. This is a high-monitor with merge capability for Original ROM. Features of both of these were converted to Upgrade ROM, and several additions made.

TO USE THE PROGRAM:

1. Copy the program listing.

2. Save it immediately, since Machine Language routines have a nasty habit of crashing, with even the slightest error.

3. 'RUN' the program, and copy down the information which appears (after about 20 seconds).

4. Save the Machine Language code with the ML Monitor if you wish to use it in the same memory location every time.

5. Activate the routine with the given 'SYS' command. Since this has been made reversible, you may de-activate with the same call. (Don't save the program when the Unwedge is active).

6. Enter 'NEW' to clear the Basic portion. You are now ready to use 'APPEND/RENUM'.

TO RENUMBER A BASIC PROGRAM IN MEMORY

1. Clear the screen, and move the cursor 3/4 of the way down the screen.

2. Enter the '<' key in the first column of a line, then hit the 'R' key.

3. If you hit 'RETURN' now, the program will be renumbered, starting at 100, in steps of 10. (The default conditions).

4. For a different starting point, simply enter that number next. Be careful that the final line number will not exceed 63999.

5. If you wish a different step size, enter a comma (','), then the new increment. Values up to 255 are acceptable.

6. After you have hit 'RETURN', the message 'RENUMBERING' will be printed. The screen memory is used for storing the line numbers, so it will now show a variety of characters.

7. When the cursor returns (5-20 seconds), the job is complete. All 'GOTO','THEN','GOSUB' and 'RUN' destinations will be changed. Any illegal or unreferenced line numbers will be numbered '65535' as a flag to the error.

8. The screen memory will handle only about 500 lines. This shouldn't be a real limitation on most programs.

TO APPEND ONE BASIC PROGRAM TO ANOTHER:

1. Place the program to be appended in Cassette #1 drive. The first line number of this program must be larger than the last one currently in memory. Use the Renumber function to prepare the program tape, if necesssary. (This is the benefit of combining these two utilities).

2. Enter the '<' key in the first column of a screen line, then the letter 'A'. (You may enter the whole word, 'APPEND', but the program only checks the first letter).

3. If you wish a specific file to be Appended from tape, enclose the file name in quotes, just as with a normal Load.

4. If there is Machine Language after the end of Basic (of program in memory), then the routine aborts with message: '?NOT ALL BASIC PROGRAM ERROR'.

5. If the program on tape is Machine Language, then this program simply Loads normally, but does not Append. The same message is printed, but without the 'ERROR'. Be careful here because the the Basic pointers will have been changed.

6. If the whole program will not fit in memory, the routine aborts. An 'OUT OF MEMORY ERROR' message will be printed.

7. The message 'APPENDING' will be printed, while the routine searches for the correct program.

8. The routine accounts for the differences between Original and Upgrade ROM. The Original ROM saved an extra byte after the end of Basic.

I hope that this proves useful for you. I am particulary indebted to Jim Butterfield, for his counsel and patience through many elementary questions.

David A. Hook
58 Steel Street
BARRIE, Ontario L4M 2E9

```
10 EA=PEEK(53)*256+PEEK(52):B=770:SA=EA
   -B:J%=SA/256:J=SA-J%*256
20 POKEEA-1,J%:POKE53,J%:POKE49,J%:
   POKEEA-2,J:POKE52,J:POKE48,J
30 J%=(SA+31)/256:J=SA+31-J%*256:POKEEA
   -3,J%:POKEEA-4,J
40 FORI=SATOEA-5:READA$:A=VAL(A$)
50 IFLEFT$(A$,1)="H"THENJ=I+VAL(MID$(A$
   ,2)):A=J/256:GOTO70
60 IFLEFT$(A$,1)="L"THENJ=I+VAL(MID$(A$
   ,2)):A=J+.05-256*INT(J/256)
70 POKEI,A:NEXT
100 PRINT"■ACTIVATE OR CANCEL ■APPEND/R
    ENUM":PRINTTAB(20)"■--WITH: ■'SYS"S
    A"■"
110 PRINT"■SAVE WITH MLM:":PRINT"■."S"
    CHR$(34)"APPEND/RENUM"CHR$(34)",01"
    ;
120 X=SA/4096:GOSUB170:X=EA/4096:GOSUB1
    70:PRINT:PRINT:END
170 PRINT",";:FORJ=1TO4:XX=X:X=(X-XX)*1
    6:IFXX>9THENXX=XX+7
180 PRINTCHR$(XX+48);:NEXTJ:RETURN
10000 DATA173,L767,H766,133,52,173,L763
      ,H762,133,53,32,121,197,162,3,181
      ,120
10010 DATA72,189,L745,H744,149,120,104,
      157,L739,H738,202,208,241,96
10015 DATA201,60,208,8,72
10020 DATA 165,119,201,0,240,8,104,201,
      58,176,239,76,125,0,32,112
10030 DATA 0,201,65,240,7,201,82,208,23
      7,76,L282,H281,162,1,134,212
10040 DATA 202,134,209,134,157,169,2,13
      3,219,32,112,0,170,240,23,201
10050 DATA 34,208,246,166,119,232,134,2
      18,32,112,0,170,240,8,201,34
10060 DATA 240,4,230,209,208,242,32,86,
      246,32,18,248,32,10,244,165
10070 DATA 209,240,8,32,148,244,208,8,7
      6,110,245,32,166,245,240,248
10080 DATA 224,1,208,235,165,150,41,16,
      208,127,162,24,173,124,2,201
10090 DATA 4,240,7,162,0,32,L128,H127,2
      40,108,32,L123,H122,56,165,42
10100 DATA 233,2,133,42,165,43,233,0,13
      3,43,160,0,177,42,240,24
10110 DATA 32,L91,H90,32,110,242,169,13
      ,32,210,255,169,63,32,210,255
10120 DATA 162,1,32,L83,H82,76,119,195,
      32,L70,H69,177,42,208,3,32
10130 DATAL63,H62,173,125,2,56,237,123,
      2,170,173,126,2,237,124,2
10140 DATA 168,165,42,56,233,2,141,123,
      2,165,43,233,0,141,124,2
10150 DATA 138,24,109,123,2,141,125,2,1
      52,109,124,2,141,126,2,197
10160 DATA 53,144,3,76,85,195,32,185,24
      3,76,221,243,32,L2,H1,230
10170 DATA 42,208,2,230,43,96,189,L11,H
      10,240,6,32,210,255,232,208
10180 DATA 245,96,13,78,79,84,32,65,76,
      76,32,66,65,83,73,67
10190 DATA 32,80,82,79,71,82,65,77,32,0
```

```
      ,13,65,80,80,69,78
10200 DATA 68,73,78,71,32,0,13,82,69,78
      ,35,77,66,69,82,73
10210 DATA 78,71,13,0,32,112,0,240,33,1
      76,249,32,115,200,72,166
10220 DATA 17,164,18,134,62,132,63,104,
      240,24,32,112,0,240,19,176
10230 DATA 249,32,115,200,166,17,134,66
      ,208,12,169,100,133,62,169,0
10240 DATA 133,63,169,10,133,66,162,36,
      32,L-115,H-116,169,254,133,33,169

10250 DATA 127,133,34,165,40,133,31,165
      ,41,133,32,32,L292,H291,160,3
10260 DATA 177,31,145,33,165,92,0,145,3
      1,136,192,1,208,242,177,31
10270 DATA 240,16,32,L301,H300,170,136,
      177,31,133,31,134,32,32,L267,H266

10280 DATA 208,220,169,255,200,145,33,2
      00,145,33,165,40,133,119,165,41
10290 DATA 133,120,208,3,32,L277,H276,3
      2,L274,H273,208,3,76,57,196,32
10300 DATAL266,H265,32,L263,H262,32,L26
      0,H259,170,240,233,162,4,221,L262
      ,H261
10310 DATA 240,5,202,208,248,240,238,16
      5,119,72,165,120,72,32,112,0
10320 DATA 176,230,32,115,200,32,L51,H5
      0,104,133,120,104,133,119,160,0
10330 DATA 162,0,189,1,1,240,15,72,32,1
      12,0,144,3,32,L82,H81
10340 DATA 104,145,119,232,208,236,32,1
      12,0,176,8,32,L102,H101,32,118
10350 DATA 0,144,248,201,44,240,192,20E
      ,175,32,L134,H133,169,0,133,33
10360 DATA 169,128,133,34,160,1,177,33,
      197,18,240,21,201,255,208,24
10370 DATA 133,95,133,94,165,94,133,96,
      162,144,56,32,85,219,76,233
10380 DATA 220,136,177,33,197,17,240,23
      6,32,L96,H95,32,L116,H115,208,212

10390 DATA 32,L62,H61,160,0,177,31,200,
      145,31,32,L90,H89,208,8,230
10400 DATA 42,208,2,230,43,136,96,164,3
      1,208,2,198,32,198,31,76
10410 DATAL-29,H-30,32,L28,H27,160,1,17
      7,33,136,145,33,32,L56,H55,240
10420 DATA 5,32,L65,H64,208,239,164,42,
      208,2,198,43,198,42,96,165
10430 DATA 42,133,31,165,43,133,32,165,
      119,133,33,165,120,133,34,96
10440 DATA 165,62,133,94,165,63,133,95,
      96,165,94,24,101,66,133,94
10450 DATA 144,2,230,95,96,165,31,197,3
      3,208,4,165,32,197,34,96
10460 DATA 32,L2,H1,230,33,208,2,230,34
      ,96,160,0,230,119,208,2
10470 DATA 230,120,177,119,96,137,138,1
      41,167,76
```

HIGH SPEED TAPE CONTROL

Wm McCraken

The NEED for HIGH SPEED ACCESS

When developing different programs for the PET it very soon becomes apparent that a great deal of time is spent in "finding" a particular program on tape, (or space to save a program).

This is due to the fact that since there is no counter on the tape recorded we cannot accurately position the tape at the start or end of any particular file.

We can "estimate" these positions using the graduations on the cassette window, (a very approximate method). We can Fast Forward or Rewind during the running of the LOAD or VERIFY PROGRAMS (again somewhat hit or miss).

This latter method, however, is the key to a considerably faster search. If we could control the Fast Forward run and know in advance the time to each file start, then we could speed up software development.

THE FAST FORWARD SPEED

As pointed out in (1), "Different types of tape and different PETs Fast Forward at slightly different speeds", and as we have all noticed or heard, the Fast Forward speed increases as the tape winds on.

These statements suggest that it is difficult to get any general rules for Fast Forwarding to discrete positions on the tape. This was the first problem I decided to tackle.

I wanted to know just how the Fast Forward time varied, so I produced a tape, "marked off", with evenly spaced timing numbers. This was accomplished using PROGRAM "PRINT ON TAPE" and a C60 cassette, (I usually use C60's).

PROGRAM "PRINT ON TAPE"

```
 10 POKE243,122:POKE244,2:OPEN1,1,1
 20 FORZ=100TO416
 30 PRINTZ:REM**PRINT ON SCREEN
 40 PRINT#1,Z:REM**PRINT ON TAPE
 50 P=PEEK(625):IFP>175THENPOKE59411,53
 60 IFP<189GOTO40
 70 POKE625,191:NEXTZ
 80 CLOSE1
```

Line 10 checks the number of characters in the tape #1 buffer and if this is greater than 175, turns on the built in cassette motor. Line 60 again checks the buffer and if over 189, shifts control to line 70 where the buffer size is made 191, thus forcing the buffer contents on to the tape, the FOR-NEXT loop is then continued.

Once the timing tape had been produced I used the following PROGRAM, "TAPE READ" in order to see at which number the tape stopped, with varying Fast Forward times.

PROGRAM "TAPE READ"

```
 10 PRINT"□TAPE READ PROGRAME"
 20 FORI=5TO100STEP5:GOSUB200
 30 OPEN1:GOSUB200
 40 GOSUB300
 50 PRINT"□":PRINT"WITH A FAST FORWARD T
    IME OF";I;"SECONDS"
 60 PRINTI:PRINT"THE NUMBER ON THE TAPE
    #1 IS"
```

```
70 FORZ=1TO10:GET#1,A$:PRINTA$;:NEXTZ
   CLOSE1:NEXT1
80 END
200 PRINTSPC(120):PRINT"PRESS REWIND ON
     TAPE #1"
210 IFPEEK(59411)<>53GOTO210
220 PRINT:PRINT"PRESS STOP ON TAPE #1 W
    HEN FULLY REWOUND
230 IFPEEK(59411)<>61GOTO230
240 PRINT"J":RETURN
300 PRINT:PRINT"PRESS FAST FORWARD ON T.
    APE #1
310 IFPEEK(59411)<>53GOTO310
320 TF=T1+60*1
330 IFTI<TFGOTO330
340 POKE519,52:POKE59411,61
350 PRINT:PRINT"PRESS STOP THEN PLAY ON
     TAPE #1
360 IFPEEK(59411)<>53GOTO360
370 RETURN
```

Line 30 - OPENs the tape channel for read
Line 70 - accepts single characters from the tape and displays them on the screen, then closes the channel.
SUB 200 - makes sure that the tape is fully rewound, and that all cassette keys are up, the key sensing being done by lines 210 and 230. Notice that the inequality sign has been used (ie <>) rather than =. This was done because when = was used, BASIC sometimes passed this control, when no switching had taken place. This was not due to switch bounce since control was held at line 210 or 230 for times of 7+ seconds. This suggests that the value of byte 59411 was at some times set to a value other than 53 or 61. I have checked this using other PETs and have found that indeed the = is unreliable. Does anyone know why this should be or is it a fault.
SUB 300 - Fast Forwards for the desired number of seconds using the built in clock.

TABLE 1

FAST FORWARD TIME (SEC)	5	10	15	20	25	30	35	40	45	50
NUMBER ON TAPE	109	120	131	143	156	170	164	214	214	230
FAST FORWARD TIME (SEC)	55	60	65	70	75	80	85	90	95	-
NUMBER ON TAPE	247	264	282	301	320	340	361	382	405	-

This was repeated many times in order to estimate the fluctuation of tape speed on Fast Forward, especially due to running cold or warm. These repeats were somewhat shorter tests, being done for time of 10, 30, 50 etc. seconds, the ranges of numbers being as shown.

TABLE 2

FAST FORWARD TIME (SEC)	10	30	50	70	90
RANGE OF NUMBERS	119-120	170-170	230-231	301-302	282-284

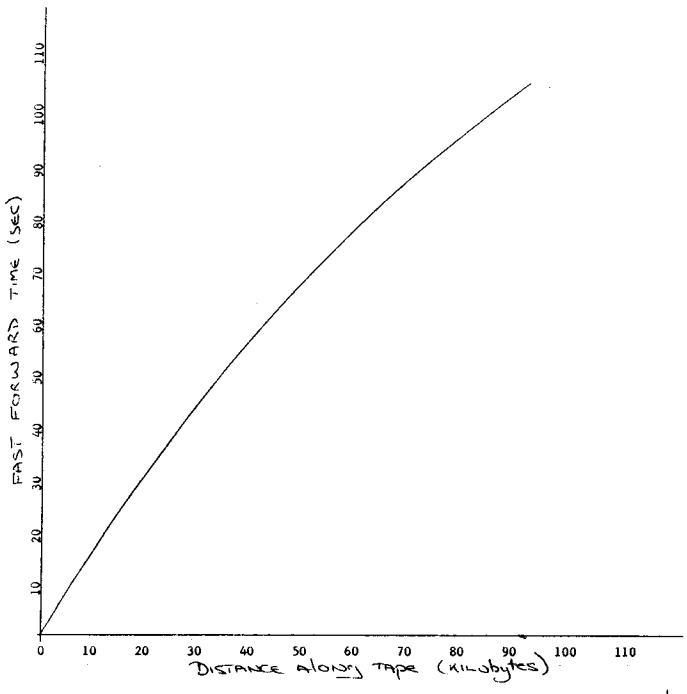
It must be remembered that each number spanned a block on the tape, the length of which was approx 5.75 sec at play Speed. This time together with other important ones was estimated using a stop watch, or the PET's clock,

Time to save 0 Bytes = 15.80 sec
Time to save 2811 Bytes = 66.35 sec
Time to save 5355 Bytes = 112.00 sec
Time to save 6678 Bytes = 135.00 sec

From this we can get the Time/Byte relationship eg
(135-15.8)/6678 = 0.018

The total PLAY time less the 7 seconds lead was measured as 31.44 minutes. This means that we are capable of storing 31.44/0.018 = 104800 Bytes on each side of the tape. Notice that the time for OPEN and CLOSE file was accounted for by the SAVE 0 Bytes time. All this information can be used to change the timing blocks into numbers of bytes.

We can now represent TABLE 1 as a graph of Fast Forward Time v Distance along tape (kilobytes).



The slope of this curve approximately halves from start to finish as we move along the X axis. This means that, as we approach the end of the tape on a Fast Forward run, the speed of the tape is TWICE what it was at thstart. This is quite significant.

The way in which this information was received was not convenient to work with, so I used a "curve fitting" package to determine the coeffecients of a polynomial which would give a good fit to this set of points. It turned out that a third order satisfied these demands.

Armed with this equation, if we know he distance along the tape, in bytes, to any

desired point we can calculate the corresponding Fast Forward time. However TABLE 2 shows that there are some fluctuations in these times and therefore prohibits a really accurate positioning.

In order that these fluctuations should have little effect it was decided that the tape should be thought of as being divided into blocks, each block being able to accept one COMPLETE memory, ie approx 7100 Bytes, plus 900 to allow for OPEN and CLOSE. This means that each side of a C60 tape can be divided into 13 equal spaces, (ie 13 programs/side).

The fact that each tape cassette has its own dynamic characteristics, can be allowed for by proportionally altering the polynomial coefficients. This assumes that each cassette's characteristics follow, fairly closely, the same pattern, a reasonable assumption.

If we therefore use the same PET with different cassetts all we have to do is establish the total Fast Forward time for each cassette and alter one line in the control program, to "tune" the system.

We can extend this thinking to allow for different PETs. All we have to do for this is to establish the ratio, PLAY time/Fast Forward time for any <u>one</u> cassette. This gives us a constant for this PET.

These ideas have been put together and are presented here as a "contents" program.

THE CONTENTS PROGRAM

I must emphasise that the reader does not have to follow all that has gone previously. The following information will give the reader a working contents program.

```
 40 REMTO BE ALTERED FOR INDIVIDUAL PETS

 50 PC=5.073E-2
 60 REMTO BE ALTERED FOR EACH CASSETTE
 65 FFT=98
 70 PRINT"□PRESS STOP ON TAPE#1"
 75 IF  PEEK(519)<>0 GOTO75
 90 GOSUB180
100 GOSUB430
110 GOSUB480
120 END
180 PRINT"□      ***CONTENTS PROGRAM***

190 PRINT"■┌─────┬──────────┬─────┬──────
    ──────┐
200 PRINT"│CODE│  PROGRAM  │CODE│  PROG
    RAM  │
210 PRINT"├─────┼──────────┼─────┼──────
    ──────┤
220 PRINT"│  -  │ CONTENTS  │  G  │
         │
230 PRINT"│     │           │     │
         │
240 PRINT"│  A  │           │  H  │
         │
250 PRINT"│     │           │     │
         │
260 PRINT"│  B  │           │  I  │
         │
270 PRINT"│     │           │     │
         │
280 PRINT"│  C  │           │  J  │
         │
290 PRINT"│     │           │     │
         │
300 PRINT"│  D  │           │  K  │
         │
310 PRINT"│     │           │     │
         │
320 PRINT"│  E  │           │  L  │
         │
330 PRINT"│     │           │     │
         │
340 PRINT"│  F  │           │     │
         │
350 PRINT"└─────┴──────────┴─────┴──────
    ──────┘
360 PRINT"■SELECT APPROPRIATE CODE FOR
370 PRINT"■POSITION ON TAPE
380 GETC$: IF C$="" GOTO380
390 IF C$<"A" OR C$>"L" GOTO380
400 BS=ASC(C$)-64: RETURN
430 REM CALCULATE FAST FORWARD SPEED
                (SECONDS)
440 BS=BS*8000
450 FT=.11594E1+.13985E-2*BS-.71234E-8
    *BS↑2+.24540E-13*BS↑3
460 FT=FT*FFT/98*19.7134*PC
470 RETURN
480 REM     ***TAPE CONTROL***
490 PRINT"□PRESS FAST FORWARD ON TAPE#1
    "
500 IF PEEK(59411)<>53 GOTO500
510 FT=TI+FT*60
520 IF TI<FT GOTO520
530 POKE519,52: POKE59411,61
540 PRINT"□PRESS STOP ON TAPE#1"
550 IF PEEK(519)<>0 GOTO550
560 PRINT"□TAPE IS NOW IN CORRECT POSIT
    ION
570 PRINT"■YOU MAY 'LOAD' OR 'SAVE' YOU
    R
580 PRINT"■PROGRAM AS NORMAL
590 RETURN
```

Line 40 - If this is the FIRST TIME the program has been run on your PET, then for any C60 cassette, determine the ratio Fast Forward time (sec)/Play Time (sec) This is PC in the program, and should be altered accordingly. This need only be done <u>ONCE</u> for any one PET.

Line 50 - Determine the Fast Forward time in seconds for the particular cassette to be used.

This is FFT in program, and should be altered accordingly. This need only be done <u>ONCE</u> for any one cassette.

Line 70 - Makes sure that all the keys on TAPE 1 are "up".

Line 190 - 470 Prints contents table and waits for code A through L, then calculates the corresponding number of blocks along the tape (BS)

Line 440 converts this number into bytes.

Line 450 calculates the Fast Forward time in seconds, using the polynomial, and allows for the length of the contents program itself (ie 3.5562 secs).

Line 460 takes account of different PETs and cassettes.

Line 480 - 590 Fast Forwards the tape to the correct position similar to PROGRAM "TAPE READ" as described earlier

HOW TO USE THE PROGRAM

1. SAVE and VERIFY contents on tape #1
2. Rewind tape#1 and LOAD/RUN
3. Tape will now be in desired position
4. Type NEW and enter your own program
5. SAVE and VERIFY your new program on tape#1
6. Update contents table

TO LOAD from this tape do steps 2 & 3 then LOAD as normal.

Note: Step 1. need only be done once for each side of the cassette.

It can be argued that lines 40, 50 and 460 can be removed from the program to make it less bother. This may be so, but the condition can then arise, due to the combination of PET and cassette, that the "block" length on the tape is considerably less than one "COMPLETE" memory. This could possibly cause "rub-out" problems.

We would eliminate this problem by making the "block" longer, (say 10 programs per side instead of 13), but this may cause longer "SEARCHING" times when using different PETs.

CONCLUSIONS

I have been using this program myself and have found that it considerably shortens the searching times Associated with LOAD and SAVE commands.

It also eliminates one problem associated with file naming, that is, all programs on the one tape could have exactly the name, since the program converts these to CODES.

We are frequently asked for information on how to use the PET to control a series of instruments for industrial use. The following article comes from Mr. R. Heath of PLESSEY TELECOMMS. LTD. and describes a very professional set up which they currently have in use. We hope that this will answer a lot of your queries.

USING THE IEEE BUS TO DRIVE INSTRUMENTS

In order to use the PET as a controller when setting up a general purpose audio test set, the standard routines described below were developed.

The basic instruments controlled were:-

1. Frequency Synthesiser.
   Hewlett Packard 3320B.

2. Digital Voltmeter.
   Hewlett Packard 34555A

3. Frequency Counter.
   Hewlett Packard 5328.

This program illustrates the operation of the three instruments under the control of the PET.

The synthesiser output is connected directly to the D.V.M. for illustration purposes and readings are displayed by the PET over the required Frequency range and at the various synthesiser output levels.

The interconnections for the set up are as shown, the IEEE ports of all instruments and the PET are parallel connected.

The frequency counter input is driven directly from the Synthesiser high level output.

```
1 REM. PET104 'AUDIO TEST'
10 PRINT"□PROGRAM TO DEMONSTRATE
   THE IEEE-488 BUS
20 PRINT"SYSTEM DRIVING A GENERAL
    PURPOSE AUDIO
30 PRINT"TEST SET.■
40 PRINT"──────────────────────
   ──────────":PRINT
50 PRINT"THE INSTRUMENTS USED ARE
   :-":PRINT
60 PRINT"HEWLETT PACKARD 3320B SY
   NTHESIZER
70 PRINT"HEWLETT PACKARD 3455A DI
   GITAL VOLTMETER
80 PRINT"HEWLETT PACKARD 5328A CO
   UNTER■"
90 PRINT"THE PROGRAM CAN BE USED
   AS A BASIS FOR
100 PRINT"A SETUP CAPABLE OF TAKI
    NG MEASUREMENTS
110 PRINT"OVER THE FREQUENCY RANG
    E 20HZ-20KHZ.
120 PRINT"IT MAY ALSO BE USED FOR
     TAKING HIGHER
130 PRINT"FREQUENCY MEASUREMENTS
    (UP TO 1MHZ)
140 PRINT"PROVIDING THE USE OF TH
    E DVM AS A LEVEL
150 PRINT"MEASUREMENT DEVICE IS A
    DEQUATE.■
160 PRINT"     PRESS A KEY TO CON
    TINUE
170 GETA$: IF A$="" GOTO170
180 PRINT"□TERMINATE THE DVM IN 7
    5 OHMS.
190 PRINT"CONNECT THE SYNTHESISER
     OUTPUT TO THE
200 PRINT"DVM AND COUNTER INPUTS.
    ■"
210 PRINT"THE PROGRAM CONVERSION
    OF VOLTS TO DBM
220 PRINT"ASSUMES A 75 OHM TERMIN
    ATION.
230 PRINT"FOR 600 OHMS, CHANGE 0.
    27386 IN LINE
240 PRINT"490 TO 0.77459
250 PRINT"THE WAIT TIMES USED REP
    RESENT 1 SECOND.
260 PRINT"THESE CAN BE CHANGED AC
    CORDING TO THE
270 PRINT"MEASURING CIRCUMSTANCES
    .■
280 PRINT"A TYPICAKL PRINT OUT OF
     RESULTS IS
290 PRINT"AS FOLLOWS:-■"
300 PRINT" SEND       SEND      R
    EC.       REC.
310 PRINT"LEV(DBM)   FREQ(HZ) LEV
    (DBM)   FREQ(HZ)
320 PRINT" -10         20        -9
    .76       19
330 PRINT" -10         40        -9
    .77       39
340 PRINT" -10         60        -9
    .77       60
350 PRINT"          ETC.■"
370 PRINT"     PRESS A KEY TO CON
    TINUE
380 GETA$: IF A$="" GOTO380
390 PRINT"□THE MEASURING RANGE OF
     THE PROGRAM AS
400 PRINT"WRITTEN IS:-■
410 PRINT"    LEVEL           -10D
    BM TO +10DBM■
420 PRINT"    FREQUENCY       20H
    Z TO 1MHZ■■■■■■■
430 PRINT"PRESS A KEY TO START ME
    ASURING
440 GETA$: IF A$="" GOTO440
450 PRINT"MEASURING"
460 OPEN9,4: REM PRINTER LISTEN
```

```
470 OPEN4,19: REM SYN. LISTEN
480 OPEN5,22: REM DVM. LISTEN OR
    TALK
490 OPEN6,25: REM CTR. LISTEN OR
    TALK
500 REM SET UP INSTRUMENT OPERATI
    NG                    CONDITION
    S
510 PRINT#4,"A+0,F800,R2,M,D3,C":
     FOR Y=1 TO 1000: NEXT
520 PRINT#5,"F2R7A1H1M3T1": FOR Y
    =1 TO 1000: NEXT
530 PRINT#6,"PF4G6S13T": FOR Y=1
    TO 1000: NEXT
540 PRINT#9,"                  FU
    LL PRINT OUT OF RESULTS.":
    PRINT#9
550 PRINT#9,"    SEND
     SEND              REC.
        REC."
560 PRINT#9,"   LEV(DBM)
    FREQ(HZ)          LEV(DBM)
      FREQ(HZ)
570 REM FOR LEVEL=-10 TO +10 STEP
     10DB
580 FOR L=-1000 TO 1000 STEP 1000

590 PRINT#4,"A"L",D2,C": FOR Y=1
    TO 1000: NEXT
600 REM FOR FREQUENCY 20HZ TO 1MH
    Z,          VALUE SET BY N AND
     DECADE SET BY P
610 FOR P=1 TO 5
620 FOR N=2 TO 10 STEP 2
630 F=INT(N*10↑P)
640 GOSUB1000
650 REM TRIGGER INSTRUMENTS AND O
    BTAIN      READINGS
660 A$="*": PRINT#5,",": INPUT#5,
    A$: IF A$="*" GOTO660
670 B$="*": INPUT#6,B$: IF B$="*"
     GOTO660
680 B=VAL(B$)
690 REM CONVERT VOLTS TO DBM
700 A=VAL(A$): A=20*(LOG(A/.27386
    )/LOG(10)): A=(INT(A*100))/10
    0
710 L1$="                    "
720 L$=STR$(L/100): L$=L$+L1$
730 F$=STR$(L): F$=F$+L1$: F$=
    LEFT$(F$,16)
740 A$=STR$(A): A$=A$+L1$: A$=
    LEFT$(A$,16)
750 B$=STR$(B): B$=B$+L1$: B$=
    LEFT$(B$,16)
760 PRINT#9,"      "L$,F$,A$,B$
770 NEXTN: NEXTP: NEXTL
780 CLOSE4: CLOSE5: CLOSE6:
    CLOSE9
790 END
5000 REM     SET SYNTHESIZER FREQU
     ENCY
5010 REM F1 = FREQUENCY
5020 REM S  = FREQUENCY RANGE SET
     TING
5030 REM D1 = AMPLITUDE LEVELING
5040 D1=0
5050 IF F>1299000 THEN S=6: F1=F
     /10000: GOTO5100
5060 IF F>129900 THEN S=5: F1=F/1
     000: GOTO5100
5070 IF F>12990 THEN S=4: F1=F/10
     0: GOTO5100
5080 IF F>1290 THEN S=3: F1=F/10:
      GOTO5100
5090 F1=F: S=2: D1=3
5100 PRINT#4,"F"F1",R"S",D"D1",C"
      : FOR Y=1  TO 1000: NEXT
5110 RETURN
```

DERIVATION OF INSTRUMENT ADDRESS
HEWLETT-PACKARD DVM (3455A)

	BINARY	PET DECIMAL	HEWLETT PACKARD ASCII
Instrument Address	10110	22	
To Listen 01	10110	54=22+32	6
To Talk 10	10110	86=22+64	V

INSTRUMENT ADDRESS SWITCHES

Address Bits 6 and 7 are set by controller

01	For LISTEN	PRINT#22,
10	FOR TALK	INPUT#22,

The above is typical of most Hewlett Packard instruments.

As written, the conversion of volts to dBs assumes 75ohms circuits are being tested, a 75ohms resistor must therefore be connected across the D.V.M. input as shown.

At each change of frequency or level it is necessary to allow a settling time for the instruments and the device being tested.

This time is variable depending upon the degree of Frequency or level change.

It is usually less than 0.2 seconds, but a wait of 1 second had been assumed in all cases for illustration.

Lines 1 to 70 describe what the program can do.

The instruments talk and listen through Files which need to be opened and then closed at the end of each test run.

Lines 190 to 198 open the required Files starting at File 4.

Note that Files 1, 2 and 3 are reserved for tape #1, tape #2 and screen respectively.

The select codes required by the PET must be in decimal, therefore the H.P. ASCII codes need to be converted as follows:

SYNTHESISER LISTEN	ASCII 3=	DEC 19 + 32
D.V.M. LISTEN	ASCII 6=	DEC 22 + 32
D.V.M. TALK	ASCII V=	DEC 22 + 64
COUNTER LISTEN	ASCII 9=	DEC 25 + 32
COUNTER TALK	ASCII Y=	DEC 25 + 64

Lines 202 to 232 set up the operating conditions for each instrument in turn, using the print command to the relevant instrument File.

The characters used are peculiar to each instrument and a full description of their meaning can be obtained from the relevant handbooks.

The meaning of the characters used in this program are as follows:

FREQUENCY SYNTHESISER

A+0	=	0dBm
F800	=	800Hz
R2	=	Range 2
M	=	Vernier out = disable Fine Frequency control
D3	=	Delay code 3
C	=	command (initiates only)

DIGITAL VOLTMETER

F2	=	Function, AC volts
R2	=	Range 7, auto
A1	=	Auto Cal., off
H1	=	High resolution, on
M3	=	Maths, off
T1	=	Trigger, internal

FREQUENCY CONVERTER

P	=	Remote program initialise
F4	=	Frequency, A input
G6	=	Frequency resolution, 1 Hz
S13	=	Multiple measurement, no service request
T	=	Reset and trigger

Line 401 sets up the loop to change the level from -10 to +10 dB in steps.

Line 410 sets the synthesiser level to the value of L and D2, C triggers the change.

Lines 425 to 440 calculate the synthesiser frequency (F).

Line 450 calls up sub 5000 which sets the synthesiser Frequency.

Values for Frequency (F1), Frequency range setting (S) and amplitude levelling (D1) are obtained.

These values are then sent to the synthesiser using the Print#4 command in line 5070.

The C in this line is the trigger command.

Line 460 sends a "comma" trigger command to the D.V.M. and loops until a new reading is obtained through input File 5.

Line 470 obtains a Frequency reading in a similar way.

Line 490 converts the voltage reading into d&m and line 500 displays the results on the V.D.U.

The program can be easily modified to allow tests to be done over any specific range of frequency and level within the limits of the test instruments.

Whilst other instruments could be substituted for those described, some alteration in setting up and trigger commands would be necessary.

The necessary alterations can usually be deduced from the relevant handbooks. However, it must be remembered that all instruments are not triggered in the same way.

TVA METER
By Bob Sparkles

This application enables the user to measure time intervals up to 6.5 seconds in units of 100 μs and to display the result on the PET screen in large letters. The measurement can be displayed with 2, 3 or 4 significant figures and a decimal point and up to four measurements can be stored internally and recalled later. This enables the user to replace the "Millisecond Timers" at present required for this purpose, and, since these cost some £200, represents a considerable saving; only a few pounds worth of additional equipment is needed.

For example, this equipment could measure the time required for a 10-cm card on a trolley to pass in front of a photo-cell. From this the velocity of the card may be calculated in the normal way. But why not let PET do the calculation automatically? The resulting velocity can then also be displayed to the whole class and we have the "Velocity Meter" for which Physics teachers have waited so long.

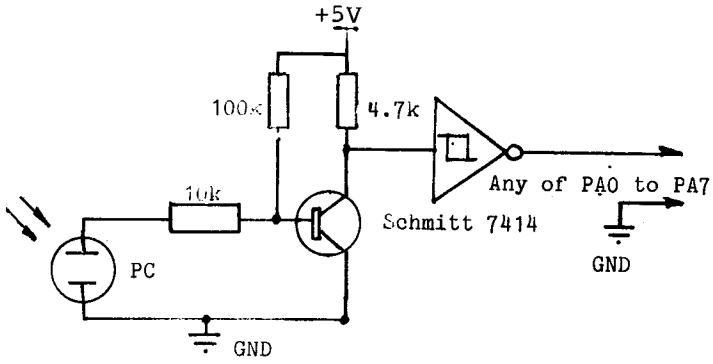
Furthermore, by using a "double" card, PET can measure the three time intervals T1, T2 and T3 and compute the acceleration from these figures.

The acceleration is given by:
A = 0.04 x (1/T3 - 1/T1)/(T2 + T1/2 + T3/2), which is derived from the familiar a = (v - u)/t.

The program is in BASIC for the calculations and in machine code for the measurements and display. The display itself is worthy of consideration, since it could easily be adapted to a display of any alpha-numeric character in large letters, and could therefore be used to communicate with the whole class at once. A discussion of this display program is given separately.

The measurement is simple. The photocell is connected to the User Port via a simple interfacing circuit and PET reads the input (decimal location 59471) and compares this with its previously recorded state. Any change in any of the inputs causes a branch to the timing mode, which continues until another change produces a stop. For acceleration three consecutive timing loops are required, so the number of loops is POKEd into a convenient location prior to the SYS command. With this system the timer can be started in one place by one photocell and stopped in a different place by another. Personally I found two photocells to be sufficient, but up to eight could be used. The inputs are also able to detect increases or decreases in the logic level, so the facility on most commercial timers of "Make to Start - Break to Stop" etc. is automatically built in.

USER PORT INTERFACE DETAILS



PROGRAM DETAILS

For those without machine code experience, the following BASIC program will load the machine code program also. This program was not planned before being written (it just grew from one idea to the next) so the expert programmer will find many instances where space may be saved.

```
5 GOTO15
10 GOSUB3000
15 W=100:U=101:L=119:H=116:K=0
20 PRINT"■TIMER, VELOCITY AND ACCELERAT
   ION METER"
40 PRINT"■FOR ACCELERATION PRESS  A"
50 PRINT"■FOR VELOCITY PRESS      V"
60 PRINT"■FOR TIME INTERVAL PRESS T"
65 GETA$:IFA$="T"THEN350
70 IFA$="V"THEN300
90 IFA$<>"A"THEN65
105 PRINT"■MEASURING ACCELERATION"
106 GOSUB1000
109 S=2:POKE251,3:SYS920
110 T1=PEEK(1001)*256+PEEK(1000)
120 T3=PEEK(1005)*256+PEEK(1004)
130 T2=PEEK(1003)*256+PEEK(1002)
140 IFT1=0ORT2=0ORT3=0THEN200
150 T2=T2+(T1+T3)/2
160 Q=4000000*(1/T3-11/T1)/T2
170 A(K)=Q
180 GOTO500
200 REM ERROR ROUTINE
210 PRINT"■I DO NOT WANT TO WAIT ALL DA
    Y!"
220 PRINT"■PRESS T TO TRY AGAIN"
223 PRINT"■PRESS M TO RECALL PREVIOUS R
    EADINGS"
225 K=0
230 GETC$:IFC$="T"GOTO15
232 IFC$<>"M"GOTO230
235 IFC$="M"ANDA$="V"THEN800
236 IFC$="M"ANDA$="A"THEN900
237 IFC$="M"ANDA$="T"THEN950
240 GOTO15
300 PRINT"■MEASURING VELOCITY"
305 GOSUB1000
309 S=1:POKE251,1:SYS(920)
310 T1=PEEK(1001)*256+PEEK(1000)
320 IFT1=0THEN200
330 Q=1000/T1:V(K)=Q
340 GOTO500
350 PRINT"■MEASURING TIME INTERVAL"
355 GOSUB1000
359 S=1:POKE251,1:SYS(920)
360 T1=PEEK(1001)*256+PEEK(1000)
370 IFT1=0THEN200
380 Q=T1*0.0001:T(K)=Q
390 GOTO500
500 REM DIGIT SEPERATION
505 PRINT"■":DP=0:SN=0
510 IFQ<0THENQ=-Q:SN=1
520 IFQ>=1THENQ=Q/10:DP=DP+1:GOTO520
540 FORI=0TO3:D=INT(Q*10)
550 Q=Q*10-D
560 IFI<DPTHEND(I)=D
570 IFI=>DPTHEND(I+1)=D
580 NEXT
590 D(DP)=10
600 IFSN=1THENGOSUB700
603 PRINT"■":FORN=0TO4:POKE251,N:POKE25
    2,D(N):SYS(662):NEXT
605 IFA$="T"THENSYS(648):GOTO660
610 SYS(634):SYS(648)
620 POKE33514,L:POKE33515,L
630 IFS=1THENPOKE33477,H:POKE33517,H
640 IFS=2THENPOKE33437,W:POKE33477,W:
    POKE33478,U:POKE33517,U:POKE33557,9
    9
660 K=K+1:IFK=FTHEN220
665 PRINT"■READY FOR NEXT READING"
670 IFA$="A"THEN109
675 IFA$="T"THEN359
680 GOTO309
700 FORI=4TO1STEP-1:D(I)=D(I-1)::NEXT
710 D(0)=11
720 RETURN
800 PRINT"■":FORK=0TOF-1:PRINT" V"(K+1)
    "  =  "VAL(LEFT$(STR$(V(K)),6)):
    NEXT
810 PRINT"■PRESS R TO RESTART"
820 GETD$:IFD$<>"R"GOTO820
830 GOTO15
900 PRINT"■":FORK=0TOF-1:PRINT"A"(K+1)"
      =  "VAL(LEFT$(STR$(A(K)),6)):
    NEXT
910 GOTO810
950 PRINT"■":FORK=0TOF-1:PRINT"T"(K+1)"
      =  "VAL(LEFT$(STR$(T(K)),6)):
    NEXT
960 GOTO810
1000 PRINT"■YOU MAY TAKE 1, 2, 3 OR 4 S
     UCCESSIVE"
1010 PRINT"■READINGS, WHICH WILL BE STO
     RED"
1020 PRINT"■AS WELL AS BEING DISPLAYED"
```

```
1030 PRINT"■WHEN YOU ARE READY TO START
     ,"
1040 PRINT"■PRESS ONE OF THE NUMBERS 1-
     4"
1050 GETB$:F=VAL(B$):IFF<1ORF>4GOTO1050

1060 PRINT"■I'M READY. LET IT GO!"
1070 RETURN
3000 REM DISPLAY ROUTINE IN MACHINE COD
     E
3005 READA:IFA<>999THENPRINT"ERROR IN D
     ATA":STOP
3010 FORI=634TO729:READX:POKEI,X:NEXT
3012 DATA999,169,13,133,252,169,178,133
     ,253,169,130,133,254,208,27
3014 DATA169,12,133,252,169,186,133,253
     ,169,130,133,254,208,13
3020 REM DETERMINE STARTING LOCATION FO
     R DISPLAY
3022 DATA165,251,10,10,10,105,24,133,25
     3,169,129,133,254
3025 REMDISPLAY REQUIRED BITS OF EACH D
     IGIT
3030 DATA162,0,165,252,10,10,10,168,185
     ,32,3,157,232,3,200,232,224,8,208,
     244
3040 DATA160,223,162,255,232,224,8,208,
     1,96,169,8,133,251,24,152,105,32,1
     68
3050 DATA169,160,200,30,232,3,144,2,145
     ,253,198,251,240,227,208,242
5000 REM BIT TABLE
5005 READA:IFA<>999THENPRINT"ERROR IN D
     ATA":STOP
5010 FORI=800TO995:READX:POKEI,X:NEXT
5020 DATA999,124,68,68,68,68,68,124,0,8
     ,8,8,8,8,8,8,0
5030 DATA124,68,4,4,124,64,124,0,124,4,
     4,124,4,4,124,0
5040 DATA64,64,64,72,124,8,8,0,124,64,6
     4,124,4,4,124,0
5050 DATA124,64,64,124,68,68,124,0
5055 DATA124,4,4,4,4,4,4,0
5060 DATA124,68,68,124,68,68,124,0,124,
     68,68,124,4,4,4,0
5070 DATA0,0,0,0,0,0,16,0,0,0,0,124,0,0
     ,0,0
5080 DATA0,0,60,32,60,4,60,0,0,0,254,14
     6,146,146,146,0
5090 DATA0,0,0,0,0,0,0,0
6000 REM TIMER ROUTINE
6010 DATA120,160,0,173,79,232,133,254,1
     73,79,232,197,254,240,249,133,254
6020 DATA169,0,133,252,133,253,173,79,2
     32,197,254,240,22
6030 DATA133,254,165,252,153,232,3,165,
     253,153,233,3,198,251,240,4,200,20
     0
6040 DATA208,223,88,96,24,165,252,105,1
     ,133,252,165,253,105,0,133,253,176
     ,7
6050 DATA162,13,202,208,253,240,205,88,
     96
6060 RETURN
```

SUB-ROUTINE FOR DIGIT DISPLAY

Each digit is printed on the screen using a 7 x 5 dot-matrix format. This is "encased" in an 8 x 8 box, which allows for three columns to be used as spacers between adjacent digits. With a screen width of 40, five digits may be displayed in one line (or four and a decimal point, or three and a decimal point and a minus sign).

Example:

Since each digit can be considered as eight rows and since each row can be stored by a single byte (one bit per column), this requires 8 bytes to store each digit used. The whole alpha-numeric system can thus be contained in about two pages of memory. Each row is represented by the decimal equivalent of the bits that have to be turned ON.

MACHINE CODE SUB-ROUTINE TO DISPLAY THE DIGITS

There are five digit positions and a number from 0 to 4 in POKEd into DIGITPOSITION (location 251) prior to the SYS command. The subroutine uses this to calculate the screen position of the top left-hand corner of the digit to be displayed. The index Y is used to move the position of each dot in position. The digit to be displayed is POKEd into DIGITVAL (location 252), from where the subroutine determines the position of the BITs by accessing the bytes stored in the table (in locations 800 to 919). A white "blank" is stored at the correct screen location for each logic 1 bit. A logic 0 bit causes a branch so that that screen location is left clear. Clearing the screen prior to a display is performed by BASIC; without this the digits could over-write each other.

The letter 'm' and 's' (for seconds, meters/second and meters/second squared) are entered by similar, less complex routines, which have not been described separately. The '-1' and '-2' on this display is handled by BASIC.

```
LINE# LOC   CODE          LINE

0001  0000                                ;************************
0002  0000                                ;* TVA METER DISPLAY
0003  0000                                ;*      ROUTINE
0004  0000                                ;*
0005  0000                                ;* BY  BOB SPARKS
0006  0000                                ;************************
0007  0000                                ;
0008  0000                                ;VARAIBLES USED
0009  0000                                ;
0010  0000             DIGVAL =165         ;NUMBER TO DISPLAY
0011  0000             BITTAB =800         ;START OF TABLE
0012  0000             TEMP   =1000        ;TEMPORARY STORE
0013  0000             BITCNT =251         ;NUMBER READ
0014  0000             SCRPOS =253         ;SCREEN POSITION
0015  0000                                ;START OF CODE
0016  0000                    *=676
0017  02A4                                ;
0018  02A4  A5 A5              LDA DIGVAL  ;GET DIGIT TO DISPLAY
0019  02A6  0A                 ASL A       ;MULTIPLY BY 8
0020  02A7  0A                 ASL A
0021  02A8  0A                 ASL A
0022  02A9  A8                 TAY         ;POINTER TO BIT TABLE
0023  02AA  B9 20 03  BITLOP   LDA BITTAB,Y  ;GET BIT
0024  02AD  9D E8 03           STA TEMP,X  ;STORE TEMPORARILY
0025  02B0  C8                 INY         ;UPDATE NEXT ROW OF BITS
0026  02B1  E8                 INX         ;UPDATE NEXT TEMP STORE
0027  02B2  E0 00              CPX #0      ;8 ROWS DONE?
0028  02B4  D0 F4              BNE BITLOP
0029  02B6  A0 E8              LDY #232    ;YES - SEND TO SCREEN
0030  02B8  A2 FF              LDX #255
0031  02BA  E8           NXROW  INX
0032  02BB  E0 08              CPX #8      ;8 ROWS DONE?
0033  02BD  D0 01              BNE BITEND
0034  02BF  60                 RTS         ;YES - FINISHED!
0035  02C0  A9 08        BITEND LDA #8
0036  02C2  85 FB              STA BITCNT
0037  02C4  18                 CLC
0038  02C5  98                 TYA         ;MOVE POINTER TO START OF
0039  02C6  69 20              ADC #32     ;   NEXT ROW
0040  02C8  A8                 TAY
0041  02C9  A9 A0              LDA #160    ;'WHITE SQUARE' CODE
0042  02CB  C8           NXBIT  INY
0043  02CC  1E E8 03           ASL TEMP,X  ;TEST MS BIT FOR 1
0044  02CF  90 02              BCC NOBIT
0045  02D1  91 FD              STA (SCRPOS),Y ;SEND 160 TO SCREEN
0046  02D3  C6 FB        NOBIT  DEC BITCNT
0047  02D5  F0 E3              BEQ NXROW   ;8 BITS DONE - NEXT ROW

0048  02D7  D0 F2              BNE NXBIT
0049  02D9                    .END

ERRORS = 0000
```

```
TVA2......PAGE 0001
LINE# LOC   CODE          LINE

0001  0000                                ;**********************
0002  0000                                ;*  USER PORT TIMER
0003  0000                                ;*      ROUTINE
0004  0000                                ;*
0005  0000                                ;*BY BOB SPARKS
0006  0000                                ;**********************
0007  0000            VIAORA =59471        ;USER PORT ADDRESS
0008  0000            STATE  =254          ;PORT STATE
0009  0000            CNTLO  =252          ;COUNTER
0010  0000            CNTHI  =253
0011  0000            BASLO  =1000         ;BASIC FETCH
0012  0000            BASHI  =1001
0013  0000            CHECK  =251          ;NO OF LOOPS NEEDED 1120  ;
0014  0000                                ;
0015  0000                                ;
0016  0000                                ;START OF PROGRAMME
0017  0000                                ;
0018  0000                  *=920
0019  0398                                ;
0020  0398                                ;
0021  0398                                ;
0022  0398  78               SEI          ;STOP PET INTERRUPTING
0023  0399  A0 00            LDY #0       ;POINT TO COSTORE
0024  039B  AD 4F E8         LDA VIAORA   ;USER PORT INPUT
0025  039E  85 FE            STA STATE    ;CURRENT STATE OF PORT
0026  03A0  AD 4F E8  WAIT   LDA VIAORA    ;WAIT UNTIL USER
0027  03A3  C5 FE            CMP STATE    ;  PORT CHANGES STATE
0028  03A5  F0 F9            BEQ WAIT
0029  03A7  85 FE            STA STATE    ;STORE NEW STATE
0030  03A9  A9 00     NEWCNT LDA #0       ;CLEAR TEMP COUNTERS
0031  03AB  85 FC            STA CNTLO
0032  03AD  85 FD            STA CNTHI
0033  03AF  AD 4F E8  LOOP   LDA VIAORA    ;CHECK FOR CHANGE
0034  03B2  C5 FE            CMP STATE    ;  OF STATE
0035  03B4  F0 16            BEQ CNTON    ;IF NOT CARRY ON TIMING
0036  03B6  85 FE            STA STATE    ;COUNT FINISHED STORE
0037  03B8  A5 FC            LDA CNTLO    ;PUT FINAL COUNT INTO
0038  03BA  99 E8 03         STA BASLO,Y  ;STORE FOR BASIC
0039  03BD  A5 FD            LDA CNTHI
0040  03BF  99 E9 03         STA BASHI,Y
0041  03C2  C6 FB            DEC CHECK
0042  03C4  F0 04            BEQ FINISH
0043  03C6  C8               INY          ;UPDATE POINTER
0044  03C7  C8               INY
0045  03C8  D0 DF            BNE NEWCNT
0046  03CA  58        FINISH CLI
0047  03CB  60               RTS
0048  03CC  18        CNTON  CLC
0049  03CD  A5 FC            LDA CNTLO    ;ADD 1 IE 100 MICRO SEC
0050  03CF  69 01            ADC #1       ;  TO COUNTER
0051  03D1  85 FC            STA CNTLO
0052  03D3  A5 FD            LDA CNTHI
0053  03D5  69 00            ADC #0
0054  03D7  85 FD            STA CNTHI
0055  03D9  B0 07            BCS ABORT


TVA2......PAGE 0002

LINE# LOC   CODE          LINE

0056  03DB  A2 0D               LDX #13
0057  03DD  CA           DELAY  DEX
0058  03DE  D0 FD               BNE DELAY
0059  03E0  F0 CD               BEQ LOOP
0060  03E2  58           ABORT  CLI
0061  03E3  60               RTS
0062  03E4                     .END

ERRORS = 0000
```

EXAM ENTRIES
James Clark

My job as a school teacher happens to involve me in organising O and A level examinations in my school, and I decided to see whether I could use my PET to store the examination entries in any useful way. (This was purely as a challenge to myself - I'm not really making any actual use of the results this year.) The O level exams provided the larger problem; I am dealing this summer with 394 candidates and 25 subjects, and I wanted as a minimum target to be able to devise a program that would output the candidate list for any given subject while holding all the data in store. At the same time, loading the program should not take prohibitively long, and altering entry details should be relatively easy. I have a standard 8K PET with no 'extras'.

This is, of course, a trivial problem if there is plenty of space available, but to fit everything into 8K requires a little trickery. I decided I didn't want to abbreviate candidates' names at all, so I concentrated on storing the subjects for which they were entered as briefly as possible. In the end I used a single integer in the range 1 to 33554431 (=2 -1) for each candidate, the integer being the sum of all the powers of 2 corresponding to the subjects for which the candidate was entered (i.e. 1st subject = 1, 2nd subject = 2, 3rd subject = 4, ..., last subject = 2). Since we have a good number of candidates only taking one or two subjects, I was able to save a lot of space by ensuring that these subject came first in the list; thus many candidates' entry code number is only a single digit.

For storage, I thought at first the best plan was going to involve blocking off the top few K of memory and POKEing my data in byte by byte. This did work all right, but I found it took rather a long time to get all the data in from tape. However, I've found that keeping all my data in data statements works very well. Loading time is not too bad (about a couple of minutes) and as I just want to make one pass right through the data per run there's no disadvantage in not being able to shift the data around easily. Space is still fairly tight, but I can fit in all the surnames (with initials where necessary to distinguish different candidates of the same name) and entries together with a self-explanatory program enabling a listing for any subject to be given. There are still about 700 bytes free, which enables small extra program segments to be written in as required (e.g. to search for candidates who have a timetable clash, or to list all those taking some specified combination of subjects).

I attach a program listing (less most of the data statements), though I dare say this is not of much general interest, being rather tailored to one particular problem.

I am going to try to fit in the exam grades as well - when they become available, but I think that I might run out of memory.

```
 1 DATANADIMI BHM,1152,NADIMI BRM,1152,D
   ORE,4,TURNER,4,BIGLAND,8
 2 DATACOLBERT,56,DUFFY PC,24,GORDON AC,
   8,HAGGIS,12,HORSTEAD,12,JOHNSON,24
 40 DATACLEMENTS,1115393,FOX,83712,HARTM
   AN,1050370,HEDDERICK,1115392
 50 DATAWILLIAMS RD,8475648,BLOM,1050371
   ,ELLIS,83201,FISHER DM,3216384
 77 DATARAYNAUD,3,STEWART PD,3,STEWART R
   MF,1,TAYLOR MJ,1,TODD,3,WALTER JF,1
 78 DATAWEBB,3,WILLOUGHBY,3,*********,33
   554431
 100 PRINT"■■ENTER THE CODE NUMBER FOR T
     HE SUBJECT TO BE LISTED:":t=20
 110 PRINT" 1 - FRENCH"TAB(T)"13 - LATIN

 120 PRINT" 2 - MATHEMATICS"TAB(T)"14 -
     GREEK
 130 PRINT" 3 - AO ENGLISH"TAB(T)"15 - G
     ERMAN
 140 PRINT" 4 - AO GERMAN"TAB(T)"16 - RU
     SSIAN
 150 PRINT" 5 - REL.STUDIES"TAB(T)"17 -
     ADD.MATHS
 160 PRINT" 6 - ART(O&C)"TAB(T)"18 - PHY
     SICS
 170 PRINT" 7 - CHEMISTRY"TAB(T)"19 - PH
     YS-WITH-CHEM
 180 PRINT" 8 - ART(LONDON)"TAB(T)"20 -
     GEN.SCIENCE
 190 PRINT" 9 - GEOGRAPHY"TAB(T)"21 - BI
     OLOGY
 200 PRINT"10 - GEOLOGY"TAB(T)"22 - MUSI
     C
 210 PRINT"11 - ENGLISH"TAB(T)"23 - DES.
     &TECH.
 220 PRINT"12 - HISTORY"TAB(T)"24 - AO F
     RENCH
 230 PRINTTAB(T)"25 - OTHERS
 240 INPUT"■■CODE NUMBER REQUIRED";C
 250 IFC<1ORC>25THENPRINT"■SUBJECT CODE
     ERROR - PLEASE REPEAT.":GOTO240
 260 C=2↑(C-1):PRINT"■";:N=0:M=0:T=0
 270 FORI=1TO395:READP$,E
 280 IF2*INT(E/(2*C))<>INT(E/C)THEN300
 290 NEXT:GOTO350
 300 N=N+1:M=M+1:IFM<21ORM=21ANDI=395)
     THENPRINTTAB(T)P$:GOTO290
 310 M=1:IFT=0THENT=20:PRINT"■"TAB(T)P$:
     GOTO290
 320 T=0:PRINT"■PRESS* TO CONTINUE LIST
 330 GETZ$:IFZ$<>"*"THEN330
 340 PRINT"■"TAB(T)P$:GOTO290
 350 PRINT"■"TAB"TOTAL NO. ="N-1
 360 IFT=0ANDM=21THENPRINT"■
 370 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■■■
```

While the control of stepping motors is not the commonest use of PET, I have been asked recently by a surprising number of people whether it is possible. Mr. Robertson's article demonstrates that it is not only possible but that the software side of the control is very neat. It puts to very good use PET's versatile language and the powerful IEEE

controller. Mr. Robertson, who has recently moved from Holland to England, also sent me details of the software which he uses to program EPROM's via the User Port. Any readers who wish to consult him more fully can contact him via PKS Designs Ltd., 40 Nuffield Rd., Poole.

IEEE 488 STEPPING MOTOR INTERFACE

Introduction

With the advent of inexpensive processing, control systems of great intelligence are possible. The IEEE-488 GPIB allows communication with many peripheral units, for all manner of supervisions and control. However, until now the bus lacked wheels and therefore could not move far! The stepping motor interface described will give it this added power thus opening up a whole new area of control.

The reason for the choice of stepping motors as the prime mover was due to PKS Designs Ltd., the manufacturer, being particularly active in this area. Also the stepping motor, being a digital machine, is well suited to digital control.

The stepping motor and drive combination has over the past few years progressed a great deal. Today these units are capable of powers in the order of 2.5 kW and shaft speeds of 3000 rpm at 1000 steps/rev. This is a far cry from the small units used in floppy discs.

Operation

Information is transferred via the bus as ASCII characters. These are compiled by the interface to give an index amount at a given rate as follows:-

1. The device address, specifying the interface.
2. The axis letter (X,Y,Z). Defining the axis.
3. The sign or motion direction. Positive sign, a space or nothing gives positive.
4. The move data. Consists of up to 6 numeric characters defining the incremental move in motor steps.
5. An ASCII "@". Used as the character to describe feedrate data.
6. The feedrate data. Consists of up to 4 numeric characters specifying velocity in multiples of 10 steps/sec.

A basic control statement may look like this:-

```
10 PRINT#5,"X-2001@102
```

This would produce a move of 2001 steps of the X axis motor in the negative direction at a feedrate of 1020 steps/sec.

Having specified a feedrate (@102) subsequent moves will also be at that rate unless another "X@" statement is given. Also an "@" statement may be given during an axis move to change the velocity without affecting the move distance.

The axis letter (X,Y,Z) defines one of three multiplexed outputs. If a simulation system is required separate interfaces for each axis drive would be needed.

The IFC control line resets the interface and if it is wished to abort an index controlling the IFC line will give that function.

Programming

A basic print statement has already been given which will initiate a move. However, how do we know if the move is complete? In the present hardwired interfaces this information is given by the SRQ line being low during index and going high on completion. The positive edge produced may be flagged inside the PET and tested by the program as follows:-

```
1060 W=PEEK(59462):POKE59427,62
1065 PRINT#5,"X-2001@102"
1070 IF(PEEK(59427)AND128)=0GOTO1070
```

Line 1060 clears the flag by reading the port and then the interrupt enable is POKEd for positive transition. The move is then programmed in 1065. 1070 tests to see if a positive SRQ transition has occured before allowing the program to continue. Line 1070 may be placed some time later if computation is required during the move time. As long as no further prints except '@' occur before line 1070 no information will be lost. Some may notice that the SRQ destination is not as described in the handbook, the values above are correct. The interrupt flag used here could of course produce a system interrupt if necessary, a machine code program for this is being considered.

Future Developments

The hardwired system is at present being implemented with an 8048. This it is hoped will also talk back when asked pertinent questions about status and axis position. This would allow parallel polling and position readout on the fly, if there is time. It would then be possible to control ten interfaces each with these three multiplexed axes. It is a big bus with 10 wheels!

Uses

As far as the PET is concerned this system would be useful in laboratory environment test systems or maybe hobbyist robots. For use in bad industrial environments the PET would probably suffer and a purpose designed

system would be better. However, it may be borne in mind that the motors can be placed tens of meters from the drives with suitable cables, thus making it possible to place the computer in protected areas.

SARTORIOUS BALANCE— PET COMPUTER INTERFACE

Peter Watts

The need for consumerprotection laws is obvious, and Britain has been a leader in this field of legislation, with many trading standards offices all around the country.

Britain is a member of the EEC however, and the EEC policy is to standardise many laws, especially in the area of trade, so as to facilitate the setting up of the proposed EEC "Freetrade" zone. It therefore became necessary to standardise the member countries' trading standards laws, and this has resulted in common 'Average weight' regulations for most of Western Europe, due to come into effect this year.

Complications have resulted however in that the new EEC system is very time-consuming to work by traditional calculations, and the need for an Automatic System has arisen.

PETALECT ELECTRONICS LIMITED are a PET main dealer who were approached by one of Europe's main balance manufacturers, SARTORIOUS, with a view to coupling their units to the PET computer. Systems were already available, but were very expensive, and the advent of computers like the PET made a lower priced system feasible.

Problems were encountered to start with because of the varying speeds of the two machines and the fact that the logic systems are different. This was overcome by development of an interface system which governs both the computer and the balance.

To keep the cost down, it was decided to use the User Port on the PET rather than the IEEE-488 bus, and a handshake routine was developed which ensured that only reliable data from the balance is transmitted.

The actual interface box itself has two push buttons on, denoted 'READ' and 'TOTAL'. When the weight of an object is required, the 'READ' button is pressed and the data is fed to the PET when the balance has stabilised. The information is then stored by the PET until the 'TOTAL' button is pressed, and is then analysed to produce the required data.

A program has also been developed which conforms to the new EEC regulations. Tradingstandard Departments in Hounslow, Berkshire and Merseyside will be using this system for their work.

Programs have been developed for use in other weights control applications, of such diverse items as biscuits, small metal products, pills, dog food and many others for manufacturers' production control department.

Work is currently progressing on a system for multiple-balance applications for use on permanent sites such as factory production lines.

These type of applications show, of course, that minicomputers such as the PET can be used in practically any application where data has to be analysed, and are particularly suited to long tedious operations, such as trading standards average weight control.

Anyone who is further interested in this system should contact PETALECT on Woking 20727.

MONITORING COST AND NUTRITION IN SCHOOL MEALS

Janet Gross-Noklaus

"As a Cook Supervisor of a girls secondar school in Cheshire, my responsibilitie include meeting cost targets an nutritional standards set by the Schoo Meals Service. In particular I have t produce meals with ingredients costing ou as closely as possible to a set figure an at the same time having a protein conten in excess of a set minimum.

These two statistics are currently compute by a central staff at Chester. I get th results 7-8 weeks after the meal has bee prepared and eaten. To be informed at thi stage that my costs are too high, o protein too low, is rather like telling of a child two months after the 'event'!.

My husband has taught me the rudiments o BASIC. He is now neck deep in machin code, driving himself, and me, mad. (I d find it just a little disconcerting to hea him shout out in bed "Shift Left", "Rotate", "Load Immediate", only to find he's talking in his sleep!!) So I decide to write a School Meals Statistics Progra on my own.

The initial requirements were to produce, each week,

1. the average ingredient cost of a school meal, broken down into four categories according to food type, and

2. the average protein content of a school meal.

The data available was:

1. My own stock sheets which show daily and weekly quantities used for each of approximately 150 ingredients; their cost per unit quantity and which food category they belong to.

2. Protein content data for each ingredient. This was kindly supplied by the Chester office, who showed some interest in my project.

The first task was to create the main file on tape. Our PET being just the basic configuration, the whole file has to be read into store before processing. The data for each ingredient is packed to save string space. The record for each item takes the form ingredient name/-category/-protein content/ (where "/" is an item separator). A sequence number, cross-referencing to stock sheets, is generated for each ingredient by the main program so is not required on the file. A glance at the old stock sheets and some experimentation showed that cost update runs would be required every week if costs were held on file, requiring an extra six minutes of data entry, processing and tape writing time. On the other hand entering costs as data on each weekly run takes less than three minutes.

Creating the file was the biggest chore. I arranged the program so that I could type from the stock sheets, with an assistant calling out protein values for each item as we came to it.

The calculations involved in the weekly run program are negligible - just a matter of accumulating totals and finding averages. The main programming effort was in formatting the input and output.

After reading the entire file, a set of headings appears on the screen

```
NO.  ITEM   AMOUNT   COST
```

then for each item the sequence number and ingredient description appear, while the cursor tabs to AMOUNT. Zero or Null 'Return' wipes out that item replacing it with the next. 'E' causes the program to go on to the next stage. Otherwise, the cursor tabs to COST. When this is entered, the total for the relevant Category is updated by AMOUNT times COST and the protein total by AMOUNT times PROTEIN.

Originally there was no provision for error correction. Once the totals were updated, the data was lost. However, there was sufficient space left to amend the program to record the AMOUNT/COST for each item, packed onto the end of the existing data in store. Two additional routines now allow for error correction. 'R' followed by the sequence (or line) number of the suspect item gets the totals 'Deupdated' and the item details displayed again for new data entry before resuming normal sequence. The other lists Line Number, Ingredient, Amount and Cost for all items processed so far, twenty per page in groups of five.

Once all ingredient data is in, the program asks for the total number of meals. When this is entered, the results are calculated - division, rounding and right align - and displayed as a table. I copy this onto the bottom of my stock sheets for a permanent record. There are only six figures, so it is no chore.

The time for a run is approximately 12 minutes for data entry and under one minute for tape reading. If I had access to the PET in the school kitchen, I could obtain daily records before the meal was served.

There are no special tricks in the program. I used my husband's home-brewed Renumber and Append programs to latch on several of his standard subroutines, such as "Data Entry", "File Write" and "Round and Align Right" which, frankly, have coding I do not understand, although I know what goes in and what comes out the other end.

Planning took possibly 3-4 hours over a two week period, the programming took me about five hours over one weekend and, with help from my husband, was debugged in another half hour. He had a bit of a giggle at some of my coding but his attitude has been - "If it works, leave it alone". I feel quite a sense of accomplishment and got a great deal of pleasure from writing the program plus a moment of ecstasy when the results of the first run tallied exactly with our laborious hand calculations.

Perhaps more importantly, my confidence in programming in BASIC has increased considerably. I feel ready to tackle something more complex. To this end I am now working on a Menu Planning Program which takes into account not only the criteria set by the School Meals Service but also what my girls like to eat!!"

USING RS232C OR V24 DEVICES WITH THE COMMODORE PET

G. C. Klintworth

The PET has captured a large proportion of the market and still offers outstanding value for money. It is particularly encouraging to hear that large quantities are being sold to Universities, schools, small businesses and other non "personal" uses, as the PET has undoubtably placed the power of a small inexpensive computer system within the reach of these institutions where their potential can be most effectively exploited.

It is inevitable that these users, as well as some home or "personal" users will wish to acquire peripherals which can be used with the PET. Such peripherals may include printers for data output and program listing, terminals for data input and output, disc storage systems for fast information access, and analogue to digital converters (ADC) and

digital to analogue converters (DAC) used to interface the PET to the analogue world. Another interesting and important application is to use the PET as a computer terminal in a large time sharing network. This puts the PET within reach of a very powerful computing facility with little additional cost.

With the exception of the ADC and DAC, the above mentioned units are normally interconnected by the EIA RS232C or CCITT V24 standards. This method entails sending and receiving digital signals in a set serial fashion, each byte framed by one stop bit, one or no parity bit, and one or two stop bits. Most common transmission speeds are 110 baud for the older systems and 300 baud for new systems. These are the speeds available on the Teletype 43, Digital Decwriter II, Texas Silent 700 series, etc. Other higher speeds are used in faster printers and non time sharing systems. Voltage levels are also specified by these standards although the range -12V to 12V is most popular, the former meaning logical 0 whilst the latter logical 1. This 24V difference between 0 and 1 together with the advantage of a 3 wire basic communication system makes the RS232C of V24 standards the ideal form of communication between single systems placed close together or far apart.

Unfortunately the PET does not have a RS232C input/output which, if one were to exist, should clearly be supported by the necessary high level software already available to the PET. As the user and memory ports are not supported by BASIC software in the same manner as the IEEE-488 bus enjoys, the designer is left with the only remaining option; the IEEE-488 bus should be interfaced to the RS232C signal lines.

With the IEEE-488 bus the user can use the following commands.

N = logical file number, M = device number.

```
PRINT#N,A,Z$,N%, etc
CMDN:LIST
CMDN:?A,Z$,N%, etc
GET#N,A$
```

The INPUT#N command has been ommitted as it does not appear to operate when using a RS232C/IEEE-488 interface; the reason is not fully understood.

These powerful BASIC commands listed above can be demonstrated by the following applications when using RS232C devices.

APPLICATIONS

1. USING A TERMINAL OR PRINTER WITH THE PET

There are many units on the market today which will still be purchased by the PET user as a result of their excellent print quality and wide paper width. Printers only have the facility to provide hard copy outputs whilst terminal units, which very often cost only a little more, have the additional advantage of containing a high quality keyboard. This latter facility allows large amounts of data such as names and addresses, stock, as well as programs to be typed into the PET.

The following statements/routines can be used with terminals and printers.

1.1 To list a program use OPENN,M:CMDN: LIST

1.2 To print data use PRINT#N,A,...,Z$ in the program

1.3 To input data use GET#N,A$

1.4 To type a program into the PET use the following routine

```
1 OPEN1,6
2 GET#1,A$:IFST<>2PRINTA$:
3 GOTO2
```

When the VDU is nearly full, STOP the program, type HOME and hit RETURN over each new statement. Hereafter type RUN CR and continue inputting the program. Upon completion type 1 CR 2 CR 3 CR to remove the inputting routine.

This procedure can be automated by the short routine listed below. Here 'cursor up' and CR are POKEd into the keyboard buffer, the line is printed onto the screen and RUN is printed beneath, when a CR from the external keyboard is sensed. This is followed by two PRINTB$ which clear the next two lines. Finally the 'cursor up' and CR's are executed by POKE525,9 after stopping the program which has the effect of placing the last typed line into BASIC memory. When a CR is passed over the line "RUN" the program commences once again.

```
PROGRAM NAME: RS232 A
  0 POKE59468,14: FOR I=0 TO 35: B$=B$+"
    ": NEXT
  1 PRINT"□";: OPEN1,6: FOR I=527 TO 532:
     POKEI,145: NEXT: POKE533,13: POKE534
    ,13
  2 GET#1,A$: IF ST<>0 GOTO2
  3 PRINTA$;: IF A$<>CHR$(13) GOTO2
  4 PRINT"RUN": PRINTB$;: POKE525,9:
    STOP
```

```
PROGRAM NAME: RS232 C
  0 POKE59468,14: FOR I=0 TO 35: B$=B$+"
    ": NEXT
  1 PRINT"□";: OPEN1,6: FOR I=623 TO 630:
     POKEI,145: NEXT: POKE631,13: POKE632
    ,13
  2 GET#1,A$: IF ST<>0 GOTO2
  3 PRINTA$;: IF A$<>CHR$(13) GOTO2
  4 PRINT"RUN": PRINTB$;: POKE158,9:
    STOP
```

2. USING THE PET AS A TERMINAL

Another very useful application is to use the PET as a terminal in a time sharing system. In the simplest example PET can be used as a dumb terminal in which case only the VDU and keyboard of the PET are used. When the intelligence of the PET is to be exploited, commands such as LIST can dump a BASIC program into the time sharing system to be stored or printed on the line printer, and data can be transferred between the two systems by the PRINT#N,A,Z$ and GET#N,A,Z$ commands.

2.1 The routine to allow the PET to run as a terminal is listed below. This program is limited to speeds of up to 300 baud due to slow PRINT command which takes 15ms to execute. If full duplex is to be used omit the statement PRINTA$; in line 3.

```
PROGRAM NAME: RS232 B

 0 OPEN1,6: POKE59468,PEEK(59468) OR 14
 1 GET#1,A$: IF ST<>2 THENPRINTA$;:
   GOTO1
 2 GETA$: IF A$="" GOTO1
 3 PRINT#1,A$;: PRINTA$;: GOTO1
 4 PRINT"RUN": PRINTB$;: POKE525,9:
   STOP
```

2.2 It is not possible to use the commands LOAD and SAVE with the time sharing system for two reasons.

2.2.1 The PET presently does not support these commands when the IEEE-488 bus is used.

2.2.2 Binary information which is non ASCII cannot be communicated. Most computer peripheral systems use the ASCII code.

2.3 PET programs can be "saved" in ASCII format by the statements OPENN,M:CMDN:LIST

2.4 A two stage routine is required if previously "saved" programs are to be "loaded" into the PET.

2.4.1 Locate the incoming ASCII file in any region of unused RAM.

2.4.2 Transfer this memory to the screen and then place CRs over each line as described in 1.4.

Remember the program stored in BASIC memory differs from the program listing displayed on the VDU; BASIC commands such as FOR, REM. etc are identified by a non ASCII code in BASIC memory.

3. USING RS232 DISC UNITS WITH THE PET

The protocol to communicate with disc units is identical in concept as those techniques described above for terminal operation. The PET becomes an intelligent terminal attached to the disc units.

IEEE—488/RS232C INTERFACES FOR THE PET

Clearly an interface should support those PET commands described earlier.

The ideal IEEE-488/RS232C interface should contain the following functions.

1. On the IEEE-488 side, subsets AH1, SH1, T4, and L2 should be supported as these subsets are used by the PET. The interface is then guaranteed to operate on the same bus as other IEEE-488 devices already connected to the PET. Note that device numbers 0 to 3 are used internally and hence devices with numbers between 4 and 15, or altogether 12 devices, can be used.

   Crystal controlled baud rates of 110 to 300 baud should be available for general terminal and printer use while 600 and 1200 baud should be available for high speed printers. The number of stop bits should be switch selectable and a reasonable range of device numbers should exist.

2. As the PET does not fully obey the ASCII character set, a comprehensive code conversion facility is required to make the PET compatible with other ASCII devices. The following code conversion is required.

2.1 All lower case letters must be converted. The PET uses the range 193 to 218 whilst ASCII requires the range 97 to 122. This conversion should occur with input and output.

2.2 The ASCII control character set is useful when using a printer eg. line feed (LF), form feed (FF), and essential when using the PET as a terminal or when using a ASCII disc system with the PET. PET only supports two of the commands viz. CR and SP. As those graphic characters which do not clash with the lower case characters are not used in other ASCII devices, they can be safely recoded as ASCII control characters. Pressing any of these keys will then transmit the proper ASCII character.

2.3 The cursor right key ⇒ , must be changed to a space.

2.4 The PET's DEL should be changed to ASCII BS.

This conversion should preferably take place within the interface as this then allows PET's BASIC commands to be used in their normal fashion.

3. A BREAK key should be included in the interface to break transmission from the time sharing computer.

4. When using the PET as a terminal to a time sharing computer systems, a time delay of typically 100ms is required for the computer to empty its input buffer. Some BASIC interpreters of other manufacturers include a NULL(I) command which appends I null characters after each CR/LF. Unfortunately this software facility is not available on the PET and hence the interface should contain such a delay function.

CONCLUSIONS

Although the IEEE-488 bus is very versatile in its function, it is unfortunate that the user must pay considerably more for peripherals and instruments containing this facility. The popular RS232C range does not suffer this cost disadvantage and it should be clear to the user that the addition of an ASCII RS232C input/output port allows the PET user to have access to a large range of RS232C peripherals which may be suited to his application.

Note: This article refers to BASIC1 machines. The restrictions described in 2.2 - 2.2.2 have been removed in subsequent versions of PET BASIC.

VIDEO OFF

Bill Williamson

Psychologists working in the field of Perception have made good use of the 8K PET's facility to swith the video screen "on" and "off". Now Bill Williamson, Chief Electronics Technician to the Department of Psychology at Leicester University, has come up with a hardware modification which provides this facility on the new 16K and 32K machines.

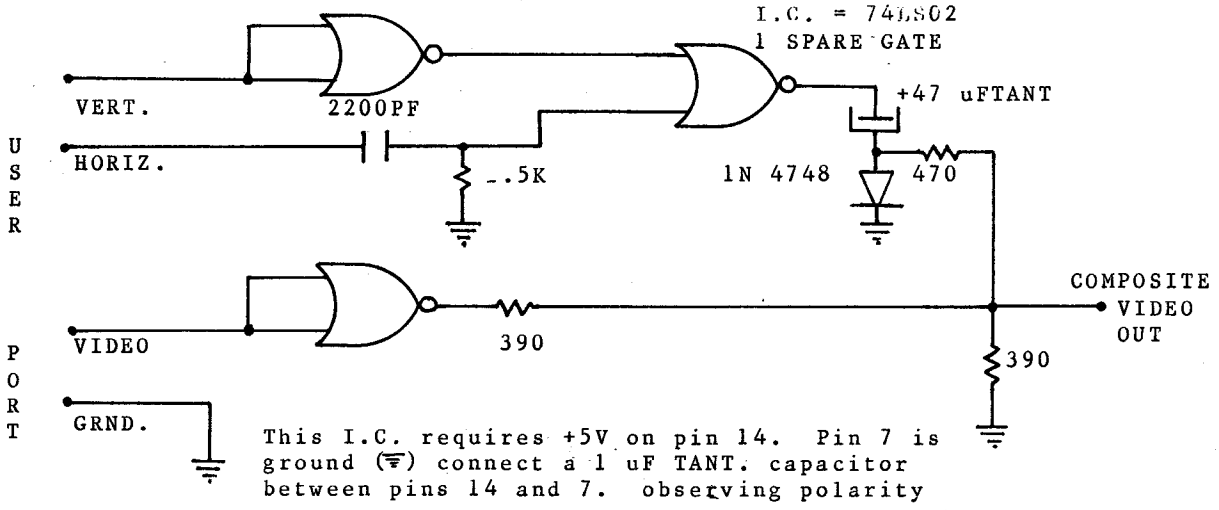
Note - Hardware modification to PET will void your warranty. This particular modification may cause unexpected side effects if your PET is used with other IEEE peripherals.

This modification allows the use of POKE 59409,52 (BLANK) and POKE 59409,60 (UNBLANK) on the new 16K and 32K PETs.

a) locate I.C. 74LS20 which is at grid reference G11 on main P.C.B.

b) cut pin 1 from the +5V rail and link it to PIN 39 on PIA#1 which is the BLANK TV OUTPUT.

Note: PIA# IS THE 6520 NEAREST THE MAINS TRANSFORMER.



ATTACHING A VIDEO MONITOR TO THE PET

Above is a simple circuit which takes the horizontal, vertical drive and video waveforms from the PET User Port and converts them to composite video suitable for driving an RF modulator or a straightforward monitor.

The circuit requires a 5 volt power supply and this may be obtained from the second cassette port which has a few milliamps available at 5V. There are no particular points to watch out for when constructing this circuit. Layout is not critical.

In the unlikely event of the horizontal hold of you display device misbehaving adjust the value of the 1.5k resistor. This will alter the horizontal sync. pulse width.

BLINKIN' LIGHTS MACHINE

The sketch overleaf may be of assistance to anyone intending to build the "Blinkin' Lights" machine described in the "User Port Cookbook" operating instructions from Commodore (Order No. MPØ31).

The whole circuit may be built on to a Prototype Circuit Board (RS Components, 488-618) which sits neatly on top of the PET, held in position by a lug attached to the rear panel top screw, and with the LED's bent forward (as shown in the diagram), enables virtually all of the procedures outlined in the "Cookbook" to be carried out with an immediate visual response, conveniently at eye level. In addition the audio amplifier connection for the shift register modes of CB2 may be accommodated on the board. It has been successfully used to produce the strains of "Auld Lang Syne" (Practical Computing, May 1979).

The LED resistors, of 330 , were chosed to limit the current drain, from Pin B-2 of J3, on the PET's power supply. With the configuration shown the total current drawn is just of 100 mA. Alternatively, Darlington Driver IC's were also tried in place of the DM7404's with a total current drain of about 60 mA for the byte display only. Ten-way ribbon cable and a small DIL Switch (RS337-560) made construction easy, only the layout is somewhat critical, with the solution indicated in the accompanying diagram. The only snags encountered were due to solder running down the pins (cut from the resistor leads) used to terminate the ten-way cable on the board. Single strand wire for jumper leads is essential, otherwise the rails on the board are easily damaged. Only typical jumper connections are shown on the diagram for the sake of clarity.

The construction and operation of the "Blinkin' Lights" machine forms an excellent introduction to understanding the intricacies of the registers of the VIA 6522 and Commodore are to be complimented for making the material available.

Some additional notes:

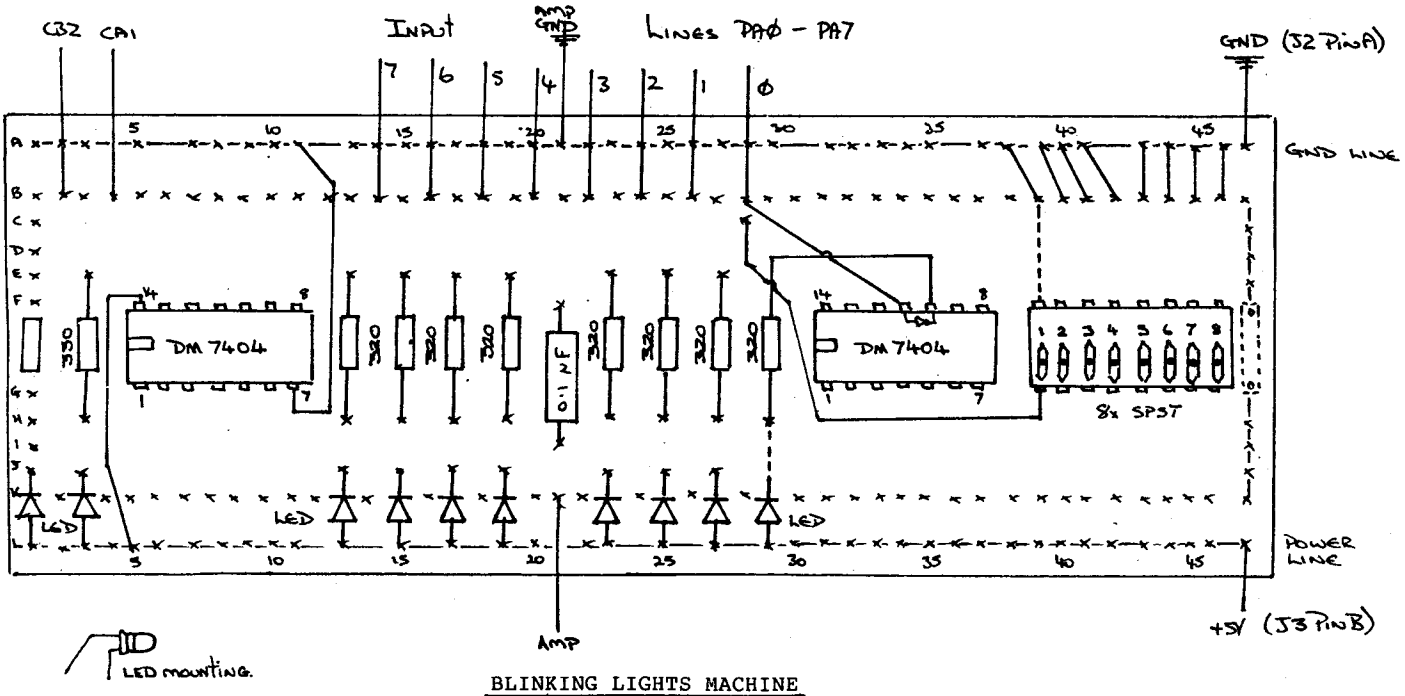
1. The prototype board contains 47 rows of 5 interconnected contacts, with continuous contact rails top and bottom.

2. The DIL Switches serve to isolate the lines - in which state the output is high with LED's ON. They also serve to ground the PAØ - PA7 lines to the low state with the LED's OFF.

3. A home-made contact of brass strip and contact pin is used in the last row of contacts to momentarily ground the CAl line.

4. The CB2 line is taken to one of the unused inverters on the 7404#2 and thence by capacitive coupling (Ø.1 .F) to the amplifier for audio output.

D. Muir
Dept of Physics
Napier College
Merchiston
Edinburgh
Tel: 031-447.7070 ext.211



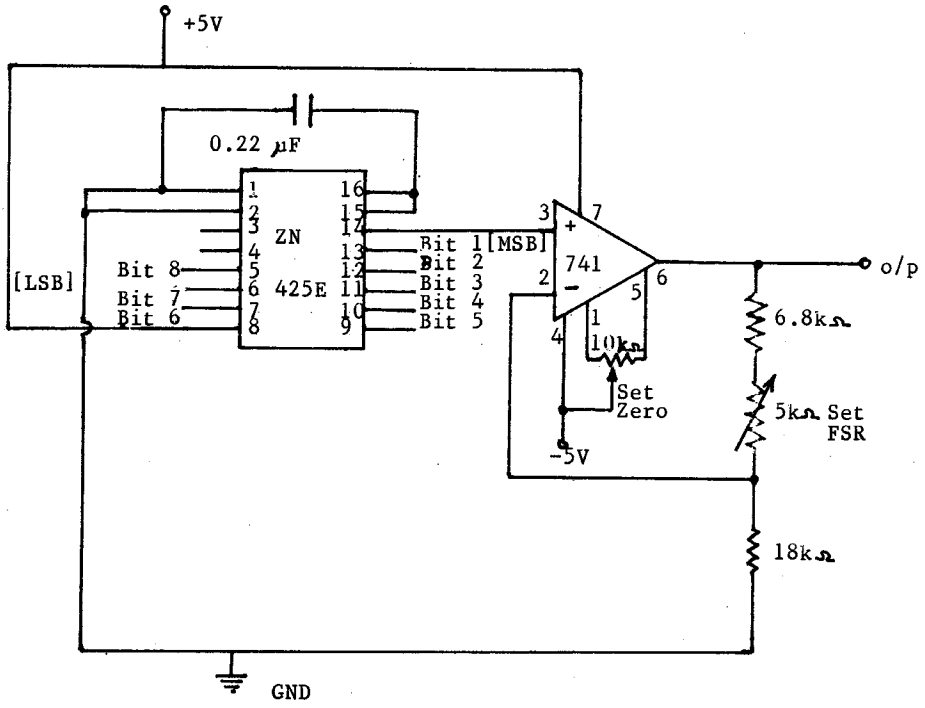
BLINKING LIGHTS MACHINE

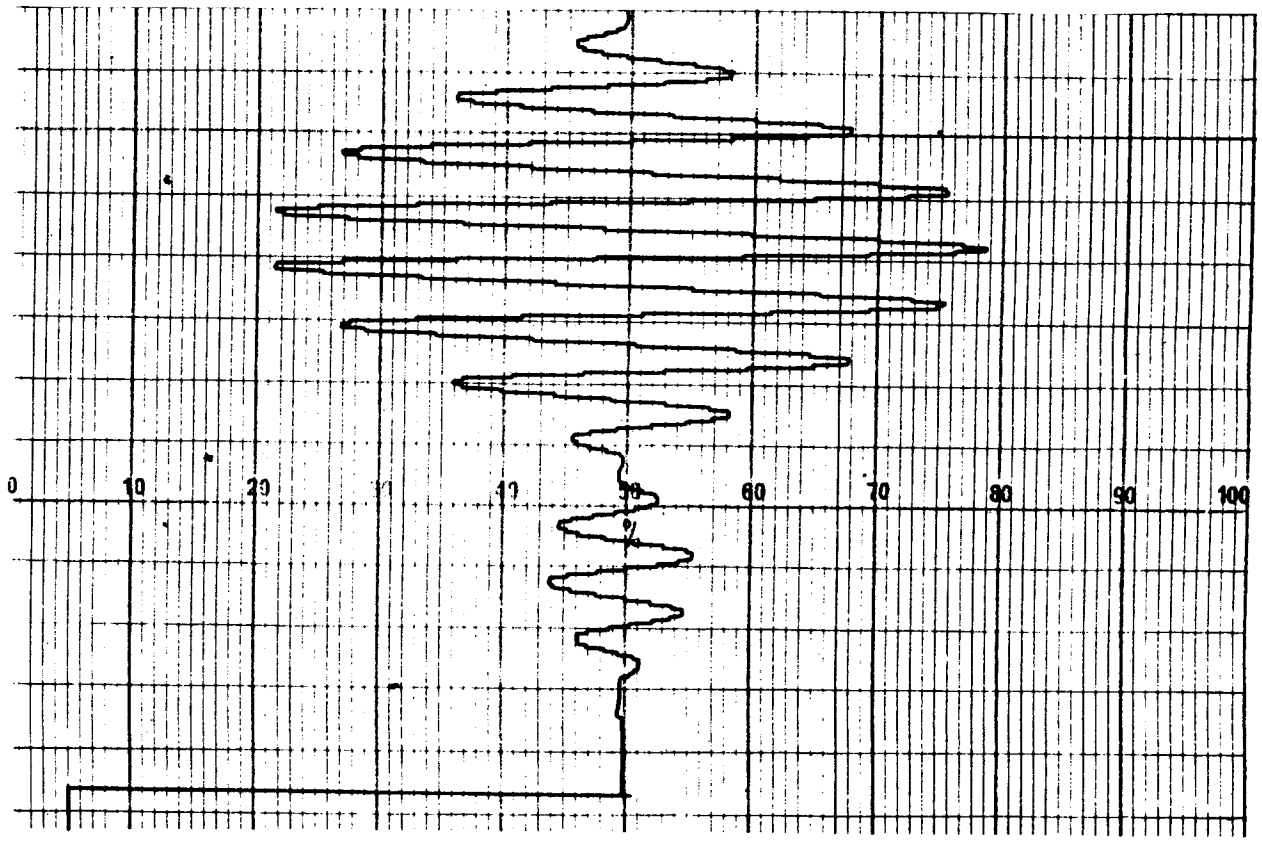
DIGITAL TO ANALOGUE CONVERSION

D. Muir of Napier College, Merchiston, Edinburgh, who sent in a design for the 'Blinkin' Lights' machine, published last issue, has sent us another circuit - this time for an analogue output from the PET.

Designed to attach to the User Port or to the 'Blinkin' Lights' machine directly, the circuit is especially suitable for driving a "servoscribe" pen-recorder as mentioned by Dr. Smyth in issue No. 4. A sample printout from this machine, along with the program to produce that waveform, is shown.

It should be noted that bit 1 on the D to A converter must be connected to the Most Significant Bit on the User Port.





HARDWARE RESET

Jim Butterfield recently demonstrated his recommended method for uncrashing a New ROM PET. This method has the supreme advantage of leaving any BASIC or machine code program intact, ie the method sucessfully by-passes the PET's inclination of destructively testing RAM as part of its 'cold start' routine.

To use the method Pin 5 (Diagnostic Sense) of the User Port must be grounded, while J4 22 (Reset) of the memory expansion bus is grounded momentarily. The ground to diagnostic sense can then be released. The PET should come up with the machine code monitor. If you wish to return to BASIC then type "X" followed by return, then type 'CLR' followed by return. Your program can then be RUN provided the house keeping variables in page zero have not be tangled up by the crash.

If you wish to stay with machine code monitor then type a semi-colon followed by return then place the cursor over the stack pointer which will have been set to '01' and over-type this with 'FA' followed by return. You can now continue with your work with the machine code monitor.

FAR INFRA-RED ASTRONOMY GROUND STATION USING THE PET WITH INTERRUPTS

A R Clark, C D Smith, I Kirk,
Leeds University Physics Department

The schematic overleaf shows the arrangement of a Ground Station to be used later this year in Texas during high-altitude balloon flights.

The slow scan video gives us the star field within an 8 x 10 field of view and the star field is updated every 2 seconds.

The 6800 synchronises to the serial data stream and decodes 32 analog words and 8 digital words. Two of the digital words give us our infrared information and the M6800 performs a handshake of 128 bytes of I.R. data every 8 seconds to the PET. Each handshake taking approximately 100 milliseconds).

Most of the other digital words define the status of our experiment and appropriate statmenets are displayed on the Kode VDU - the status being upgraded every 4 seconds.

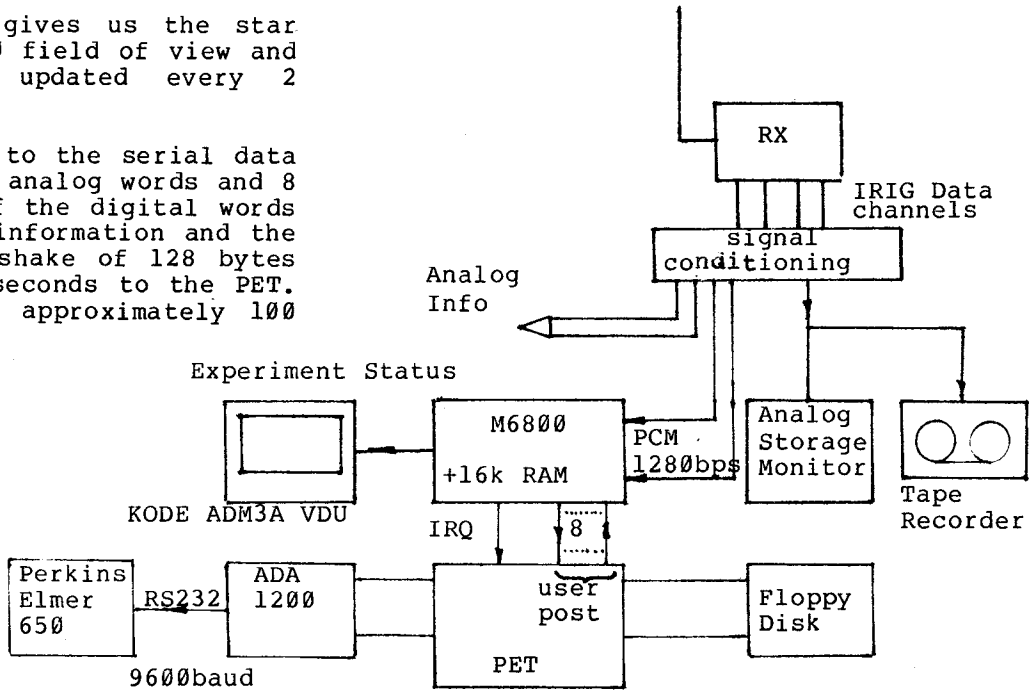
The infrared data is displayed graphically on the PET screen, partially processed and then stored on the Floppy. As the printer takes approximately 15 seconds to print out a full page, we can obtain a hard copy every 3 handshakes.

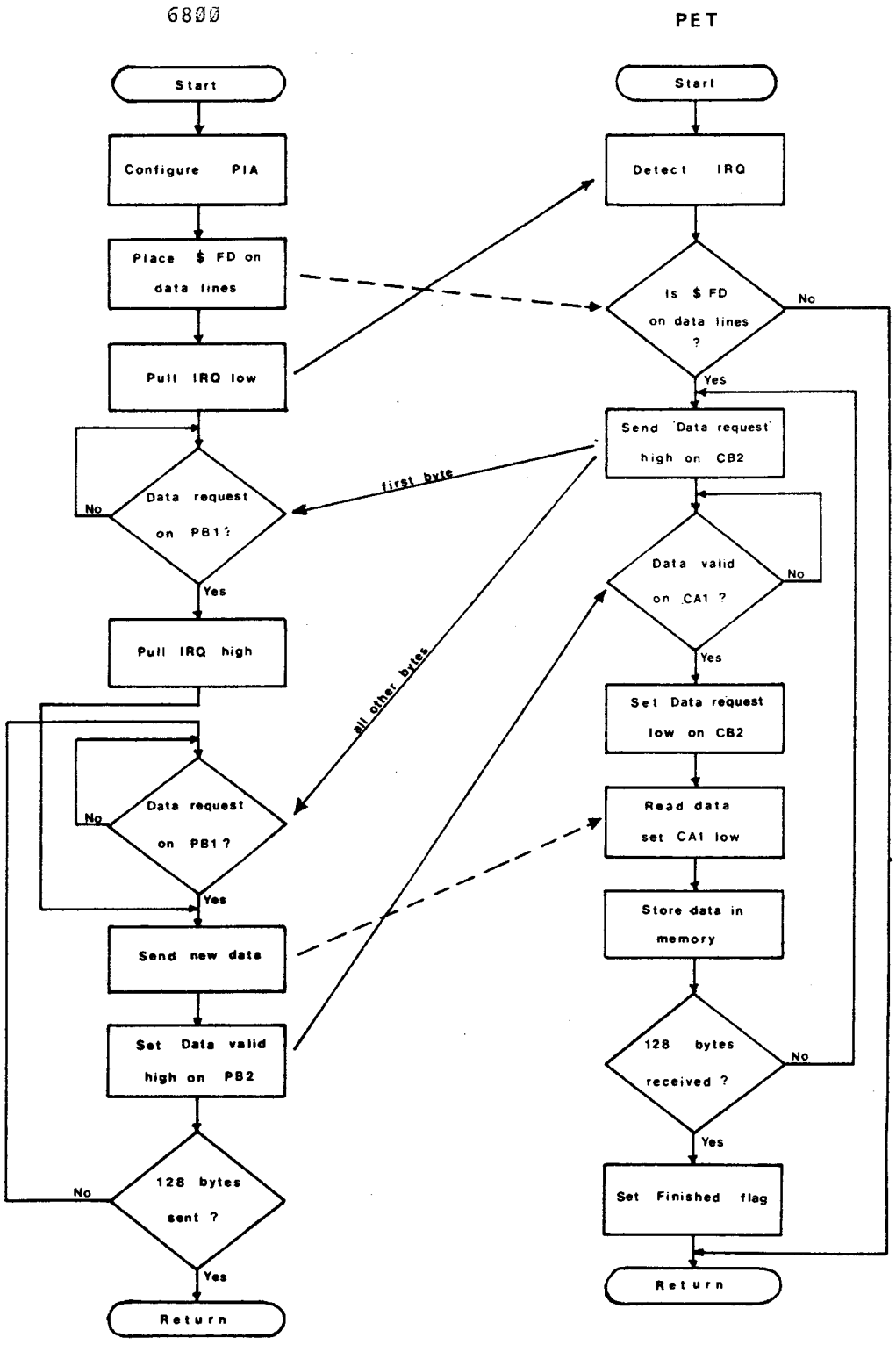
The following notes give further details of our technique for handshaking data between the PET and external microprocessor.

PET - M6800 DATA HANDSHAKE

Figure 1 shows the basic flow diagram for the handshaking of 128 bytes of data from a Motorola M6800 into the PET. The data arrives from the M6800 onto the user port data lines (memory location $E841), and the interrupt request comes to the memory expansion port which is also connected to a floppy disk. The handshaking routine is designed so that this operation is carried out before the PET services any of its internally-generated interrupts, and resides in the cassette #2 buffer - starting address $033A (826 decimal). The machine code routine is shown in Figure 2.

In order that the above routine is executed before any internal interrupts, the vector pointer must be reset so that PET jumps directly to $033A. This could be done directly by the commands: POKE 537, (low order byte): POKE 538, (high order byte), but if an interrupt occurs before the high order byte has been stored, the system will crash. For this pointer must be reset by the machine code





M6800/PET Data Handshaking

routine shown in Figure 3, which is loaded from bASIC into the top of cassette #2 buffer, and run by a SYS command. If PET is servicing a routine when the M6800 sends IRQ will continue with that routine, but IRQ will still be low when it finishes, so PET will service M6800 immediatelly on completion of the RTI instruction. Hence the M6800 will continue to request until PET acknowledges by sending the 'DATA REQUEST' pulse.

(The handshake routine finishes by storing a flag in $0377 (887 decimal) which can be tested from BASIC by the PEEK(887) command, always remembering to reset the flag as soon as it has been found).

```
033A  78          INTEN   SEI          :re-enable system interrupt
      A9 85               LDA $85      :low order byte of vector
      8D 19 02            STA $0219
      A9 E6               LDA $E6      :high order byte of vector
      8D 1A 02            STA $021A
      60                  RTS
0347  78                  SEI          :disable system interrupt
      A9 5C               LDA $5C      :low order byte of vector
      8D 19 02            STA $0219
      A9 03               LDA $03      :high order byte of vector
      8D 1A 02            STA $021A
      A9 7F               LDA $7F      :low order top of memory
      85 86               STA,z $86    :pointer location 134
      A9 1F               LDA $1F      :high order top of memory
      85 87               STA Z $87    :pointer location 135
      58                  CLI
      60                  RTS
035C  AD 41 E8    DATAIN  LDA $E841    :data handshake routine
      C9 FD               CMP $FD      :start read data lines
      D0 29               BNE END      :if not $ FD goto END
      A0 00               LDY $0       :set index to zero
      A9 E1       DATA1   LDA $E1      :if FD on data lines set
      0D 4C E8            ORA $E84C    :CB2-data request-high
      8D 4C E8            STA $E84C
      A9 02       DATA2   LDA $02      :wait for data valid on CA1
      2D 4D E8            AND $E84D
      F0 F9               BEQ DATA1    :if not goto DATA1
      A9 DF               LDA $DF
      2D 4C E8            AND $E84C    :pull CB2 low and remove
      8D 4C E8            STA $E84C    :data request
      AD 41 E8            LDA $E841    :input data and store in top
      99 80 1F            STA,Y $1F80:of memory using index pointer
      C8                  INY          :increment index
      C0 80               CPY $80      :is index = 128 if not then
      D0 DE               BNE DATA1    :goto DATA1
      A9 FF               LDA $FF      :set flag to 255 and store
      8D 8F 03            STA $038F    :in location 911
      4C 85 E6    END     JMP $E685    :jump back to Basic
038F  :               END FLAG POINTER
```

Basic Programming

A SHORT NOTE ON MOVING THINGS

Difficulty has been experienced by some of our users in moving the cursor under programme control and questions have been asked about how graphs and plots such as sine curves can be displayed.

As you probably already know, cursor control characters in quotes when printed will cause the cursor to move accordingly. If your experiments so far with this technique are giving slightly odd results, do make sure that you are terminating your print statement with a semi-colon. Failure to do this will cause the machine to output a carriage return/line feed at the end of the print statement, leaving the cursor just where you don't want it.

ARE YOU READY?

There have been reported mysterious occurences of the out of data error when editing and fiddling about in general.

This is not a bug, but is due to pressing RETURN whilst the cursor is over the READY prompt. The machine interprets this as READ Y and as there is usuallly no corresponding data statement around we get the error.

EDITING

There is an interesting property of the screen edit routine which gives rise to the followinng effects :

If you insert using the INS key, more spaces than you type in characters, the DEL key must be pressed twice the number of times there are spare spaces. E.g. If you insert six spaces in a middle of a line and only type in four new characters, the first two presses of the DEL key will produce inverse characters which will disappear on the next two presses. Remember, the INS key will move all characters including the one under the cursor to the right, whilst the DEL key will delete the character on its immediate left.

DELAYS

Quite a few people have asked how to put delays into programs. Here are two common methods :

For a delay of approximatly 1 second:-

```
10 FOR I=1 TO 1000: NEXT I
```

This will cause a delay of approximately 2 seconds.

```
20 FOR I=1 TO 2000: NEXT I
```

The following routine makes use of the PET's real time clock. There is a command called TI which always contains the number of 'jiffies' since the PET was first switched on

```
10 T=TI
20 IF T-TI<60 GOTO20
```

Lines 10 and 20 cause a delay of approximately one second and work as follows:-
Line 10 sets the variable T equal to the real time jiffy clock TI (a jiffy is 1/60 of a second)
Line 20 tests to see whether 60/60 of a second have elapsed, if not the program returns to the beginning of line 20 and checks again.

Here is a small program you might like to try which uses delays involving the real time clock in an interesting manner.

```
$0$0PROGRAM NAME: TIME PROGRAM

     5 PRINT"KEY IN A NUMBER: ";
     10 T=0: A$=""
     20 GET K$: IF K$="" GOTO20
     30 T=TI: GOTO60
     40 GET K$
     50 IF TI-T>60 GOTO70
     60 IF K$<>"" THEN PRINTK$;: A$=A$+K$:
        T=TI
     65 GOTO40
     70 IF A=0 THENPRINT" + ";: A=VAL(A$):
        GOTO10
     80 PRINT" ="A+VAL(A$)
```

PLOTTING

On the subject of plotting and layouts, the basic trick used with this type of graphic system is to first create strings full of cursor movement characters and then access parts of them using LEFT$, etc. dynamically.

```
PROGRAM NAME: PLOT - STRING

     10 Y$="[inverse cursor-control characters]"
     20 X$="[inverse cursor-control characters]
        [inverse cursor-control characters]
     30 INPUT"[inverse character]INPUT X,Y";X,Y
     40 PRINTLEFT$(Y$,ABS(Y+1));LEFT$(X$,
        ABS(X));"*"
     50 GOTO30
```

It is possible, with very little effort, to address locations on the screen directly using simple XY co-ordinates. Below we have a simple program that enables one to do this.

PROGRAM NAME: PLOT - BY POKE

```
10 INPUT "■INPUT X,Y CO-ORD";X,Y
20 IFX>40ORY>25GOTO10
30 POKE32768+X+Y*40,42: GOTO10
```

The line that does the actual XY co-ordinate conversion is line 30. X has a range of 0-39 and Y has a range of 0-24.

REMARKS

When writing REMark statements, graphics and lower case can be included if they are put inside inverted comma's. This enables separating lines such as:

```
10 REM"_____"
```

Try writing a line of graphics characters next to the REM without the inverted commas and see what happens.

ARRAYS

When using subscripted variables such as A(4) the operating system automatically reserves 10 elements without having to declare a dimension with DIM. If, however, you are using a very long program and are using less than 10 elements per variable - say 4 - it will save space to declare the dimension's length. For example:

```
10 DIM A(4),B(5),B$(2)
```

NUMBERS

To display a number (N) to D decimal places, use the following routine:

```
10 INPUT"PLACES OF DECIMAL■■■■■";A$:D
   =VAL(A$)
20 INPUT"NUMBER■■■■■■■";A$:N=VAL(A$):
   IFA$="*"THENEND
30 PRINTINT(N*10↑D+0.5)/10↑D:GOTO20
```

ONE LINER

For an intriguing display of graphics, try running this one line program entitled "BURROW"

```
1 A$="■■■■■":PRINTMID$(A$,RND(.5)*4+1,1)
  "*■■";:FORT=1TO30:NEXT:PRINT"■■■";:
  GOTO1
```

80th CHARACTER

When trying to get the maximum number of characters onto a line of BASIC, it can be frustrating to find that the last character (80th) cannot be entered since the cursor will then move onto the next line. However, it is possible to "push" a character, by inserting, along the line to this end position.

INPUTTING

When using INPUT remember that the interpreter will take the whole contents of the line and not just the characters which have been input. Thus it is possible to get errors when making a numeric input when there is another character on the line. The errors which will normally be given are REDO FROM START or EXTRA IGNORED.
To get around this it is a good idea to use string input and then change this into numeric input for actual use within the programme. ie.

```
10 INPUT"■INPUT A NUMBER■■■■■■■";A$
20 IFA$="*"GOTO10
30 A=VAL(A$): PRINT"■THE NUMBER IS";A
40 GOTO10
```

Line 10 accepts a string from the keyboard. Note also that it has been made crash proof by a simple bit of cursor control. The following will be printed to the screen:-
INPUT A NUMBER? *
If return is pressed then the program will accept the "*".
Line 20 checks to see if a "*" has been input, if it has then the program jumps back to line 10 again.
Line 30 turns A$ into the numeric variable A. A will be set to 0 if a string input is made.

The following routines were developed by Paul Higginbottom for use in the Commodore PET Centre.

The first, which is for 8K PETs but can be adapted for new ROMs, allows repeated cursor movements. The cursor in this routine is shown as an asterisk and will move in the desired direction as long as the key is pressed down:

```
10 PRINT"■■■■■■■■■■■";
20 A=(PEEK(515))OR(PEEK(516)*128)
30 IFA=255ORA(<>66ANDA<>73ANDA<>194ANDA
   <>201)THEN20
40 IFA=66THENPRINT"■■■■■";:GOTO20
50 IFA=73THENPRINT"*■■";:GOTO20
60 IFA=194THENPRINT"■■*■■";:GOTO20
70 IFA=201 THENPRINT"■■■*■■";:GOTO20
```

The second will produce a "perfect resolution" sine wave down the paper on a Commodore 3022 tractor feed printer:

```
10 DIMA$(5)
20 FORI=0TO5
30 FORJ=0TO5
40 READA:A$(I)=A$(I)+CHR$(A)
50 NEXTJ,I
60 DATA64,0,0,0,0,0
65 DATA0,64,0,0,0,0
70 DATA0,0,64,0,0,0
75 DATA0,0,0,64,0,0
80 DATA0,0,0,0,64,0
85 DATA0,0,0,0,0,64
100 FORI=0TO2*πSTEPπ/100
105 B=(SIN(I)+1)*230
110 OPEN6,4,6:PRINT#6,CHR$(5)
120 OPEN5,4,5:PRINT#5,A$((B/6-INT(B/6))
    *6)
130 OPEN4,4,0:PRINT#4,TAB(INT(B/6));
    CHR$(254)
140 CLOSE4:CLOSE5:CLOSE6:NEXTI
```

TIMING TABLES

BASIC STATEMENTS

CONSTRUCT	APPROX TIME (MILLISEC)
FRE	1 to 10
PEEK,POKE	1
TI$	3 to 4
TI	1
GET	1 to infinity
POS	1
PRINT X or	
PRINT	15 to 19
PRINT X$;	14 + LEN (X$)/2
READ X and	
DATA 3	9
REM	0.2 to 2
RESTORE	0.3
TAB	2
SPC(N)	1 + 0.6*N
FOR 1 = ...	
NEXT 1	4.0 + (1.6 each)
STEP	1.3
IF	0.4
GOTO or GOSUB	1.1
ON A GOTO or GOSUB	
L L	0.5 + (0.3*A) + (0.2*M)
RETURN	0.9

Using colon,:, saves 0.6 over new line.
SAVE or LOAD
15 sec + (2 sec per 100 char)
i.e. 500 baud.

STRING FUNCTIONS

FUNCTION	APPROX TIME (MILLISEC)
+	0.5 + (0.2 per char)
ASC	1
CHR$	1.2
LEFT$, RIGHT$	3 + (0.025 per char)
LEN	0 to 8
MID$	4 + (0.025 per char)
STR$	7 to 10
VAL	1.3
=,<>,<=,>,>=	3 to 4

ARITHMETIC FUNCTIONS

FUNCTION	APPROX TIME (MILLISEC)
ABS	0.6
ATN	42
COS	27
EXP	27
INT	1.2
LOG	23
RND RND (-1)	1.0
RND (0)	0.9
RND (1)	4.1
SGN	1.1
SIN	25
TAN	50
user FN	2.4

ARITHMETIC OPERATORS

SYMBOL	APPROX TIME (MILLISEC)
0 B,1 B	0.3
2 B	32
else	50 to 100
/ O/B,A/1	0.5
else	2 to 5
* O*B, A*O	0.4
else	1.5 to 3
+	0.3 to 1
-	0.3 to 1
=,<>,<,<=,>,>=	0.7
AND, OR	1.7
NOT	1.4

VARIABLES AND CONSTANTS

ITEM	APPROX TIME (MILLISEC)
A,A$,A=,A$=	0.7 to (0.7 + nv*0.1) nv = no. of variables in program
AA,AA$,AA=,AA$=	0.2 more than above
A%	0.3 more than above
A%=	0.6 more than above
999	1 per digit
.999	0.7 + (4.2 per digit)
E16	0.2 + (0.4* exponent)
E-16	0.2 + (3.0* exponent)
"ABCDE"	(0.6 to 0.7) + 0.02 per char)
M (I,J....)	(1 to 1.5)* (no. of subscripts)

TIMING PROGRAM

```
10 REM**********TIMING PROGRAM**********
100 N=300
110 T1=TI
120 FOR I=1 TO N
130 REM PUT TEST CONSTRUCT HERE
140 NEXT I
150 T2=TI
160 FOR I=1 TO N
170 NEXT I
180 T3=TI
190 PRINT"■■■■■■■■■■■■■■■TIME:  "1000*((T
    2-T1)-(T3-T2))/60/N
200 LIST
```

MEMORY USAGE (IN BYTES)

BASIC 1028 (1/0 buffers, tables etc)
each statement
- 4 for line number and following space, regardless for the line number
- 1 for each BASIC keyword
- 1 for each character, including RETURN

each variable with a value assigned, regardless of spelling or value takes 7 bytes; for string variables, add the length of the string

each array (N.B., size includes 0th element) take f* (size + 1) + (2 per dimension) where f=5 for floating point arrays, f=2 for integer arrays, and f=3 for string arrays.

The system slows down noticeably when memory is nearly full.

BEGINNING BASIC

STRING HANDLING IN BASIC LANGUAGE

Most people are aware of a computer's calculating abilities. However, there is another side to computing that is not as familiar to the average person. We have all been exposed to computerised mailing lists, credit information, airline ticket reservations or some other example of the computer data handling capability. The purpose of this article is to relate how some of these things can be accomplished even using a microcomputer such as the Commodore PET.

Most microcomputers today come with BASIC as the major language. BASIC was designed to be a very simple language, easily learned, which can handle most of the operations that are associated with large computers. The penalising aspect of BASIC is the speed of execution which is often not a major consideration in the use of fixed handling data. We will try to explain each and include examples.

Before we begin the discussion of the commands of BASIC, it is necessary to define some terms. We must undertand how the computer stores its data as code numbers. The microcomputer is acutally only capable of storing numbers in the range 0 to 255. As a result of this some standard codes have been created. The code convention used by micros is called ASCII (American Standard for Computer Interchange of Information). For example, in ASCII code the letter "A" is represented by the number 65, "B" by 66, and so forth. The computer contains a decoding routine that automatically decodes the 65 and prints out an "A". Numbers can be represented in three ways inside the computer. The first is as a literal string using the ASCII representation in each number. No arithmetic can be done on numbers represented this way. The most common representation is called "floating point". The computer allocates a number of bytes or memory locations (usually five or six) to represent a number in a way which makes all the arithmetic possible. The last representation is called "Integer" where the range of the number is restricted to +/- 32767 as rounded integers. The only important part is that most micros handle data, either numeric or alphabetic, as the first classification - literal strings of ASCII characters. Since some arithmetic process is usually necessary, there are techniques for converting between the various data representations.

BASIC AND STRING HANDLING

A "string" is represented in BASIC by a "$" following the variable name. For example, A$, IN$, or X1$. A string is a series of ASCII coded characters stored together in a sequence in memory ("A$=ABCDE"). If we looked where the computer had stored A$, we would see 65,66,67,68,69 in adjacent memory locations. Note that in the example above the quotations marks indicate to the computer to store the ASCII values of the letters between them and are therefore mandatory.

Strings of ASCII characters can be manipulated by using a series of commands. The first are LEFT$, RIGHT$, and MID$. These commands allow us to view portions of strings and are used extensively in the retrieval and evaluation of data stored in the computer.

LEFT$
Example: A$=LEFT$("ABCDE,2)
Print A$ will result in AB

The string to be manipulated is entered as a literal (as above in quotes) or as a variable (X$) followed by a comma and a number. The number designates how many characters starting at the left will be extracted from the main string.

RIGHT$
Example: A$=RIGHT$("ABCDE",2)
Print A$ will result in DE

Obviously, RIGHT$ accomplishes the identical task as LEFT$ but the operation is performed on the other side.

MID$
Example: A$=MID$("ABCDE",2,2)
Print A$ will result in BC

In this case, the first number following the string value is a pointer to the first character to be extracted, starting from the first character on the left (character A is the first, B the second etc). The next number designates the number of characters to be extracted.

A$=MID("ABCDE",2) this form of the MID$

command allows the user to remove the leftmost character ie
A$="CDE"

In addition to the above string handling commands there are another three instructions which allow manipulation or study of strings.

LEN
Example: A=LEN("ABCDE)
Print A will result in 5

LEN returns the length of a string and is used extensively in setting up data in files. We will see more of LEN later.

VAL
General for A=VAL(124.35)
Print A will result in 124.35

VAL is used to convert literal numbers (as ASCII strings) to their "floating point" form so that arithmetic can be performed.

It is used widely for doing calculations on information stored in data files.

STR$
Example: A$=STR$(A):where A=124.35
Print LEN(A$) will result in 7

STR$ is the exact reverse of VAL. It is used to return calculated values to ASCII string form for storage in data files. An important note is that STR$ always leaves a space at the beginning of the string representation of the number for the sign (+ or -). If the number is positive it is preceded by a space or if it is negative by a minus sign.

The last two string handling commands are normally used in advanced programming to reduce storage requirements to the absolute minimum.

ASC(A$)
Example: A=ASC("A")
Print A will result in 65
This command converts a character to its ASCII code and allows it to be used in calculations.

CHR$(A)
Example: A$=CHR$(65)
Print A$ will result in A

This is the reverse of ASC and uses the computer's coding logic to generate the appropriate character from a given code.

STRING HANDLING OPERATORS

The following operators can be used alone and in combination with the above commands and information to allow very powerful decision making programs using string data. The operators are <,>,= and +.

Since strings are represented as numbers in the computer, they can be handled similarly in many respects. The ASCII code of both numbers ("2") and letters ascend with ascending values of alphabetic order. This allows us to compare apparent magnitudes of strings. It is true that the ASCII code of B(66) is greater than (>) A(65) so we can say that B$>A$ or A$<B$ or A$<>B$. All of these tests would be true. This allows us to compare ASCII and place it in alphabetical or numerical order using simple comparison operators. The only difference in the use of these operators for numeric or string information is in the use of (+).

For example: A$="ABC"+"DE"
Print A will result in ABCDE

1. A SUB-ROUTINE LIBRARY

Mike Gross-Niklaus

I copy every useful routine I come across or develop into an indexed notebook. This saves me time and avoids errors when writing the standard sections of my programmes. The 'Templeton' merge, described in the last issue, cuts out the error-prone business of keying in long routines, and makes the compilation of an indexed subroutine library an attractive proposition. Whether or not you use the merge, I strongly recommend that you create an indexed library of your favourite routines.

To add to your library, here are some which have proved useful to me. The shorter ones don't use the merge efficiently, and are best keyed in directly from your notebook.

1.1 Right Align.

This is intended mainly for cash columns. Given a value to two places of decimals in Z, then Z$ (9 chrs) is returned with the value aligned to the right, zero appearing as a single nought.

```
55000 REM  --- CASH RIGHT ALIGN ---
55010 Z$=LEFT$(RIGHT$("        "+STR$
      (Z)+.005*SGN(Z)),10),9):RETURN
```

1.2 Input Trap.

This INPUT routine copes with a null return and with commas and also allows editing prior to pressing RETURN. It uses the keyboard buffer to type two quotes followed by a delete after the INPUT question mark.

```
55100 REM --- INPUT TRAP OLD ROM ---
55110 POKE525,3:POKE527,34:POKE528,34
      :POKE529,20:INPUTA$:A=VAL(A$)
      :RETURN

55150 REM --- INPUT TRAP NEW ROM ---
55160 POKE158,3:POKE623,34:POKE624,34
      :POKE625,20:INPUTA$:A=VAL(A$)
      :RETURN
```

1.3 Detect SHIFT key up or down.

Adapted from a routine shown me by Andrew Lister working at the Manchester College of Higher Education, this is a neat way of getting two operations out of one key, one when it's down and another when it's up.

```
55200 REM -- DETECT SHIFT KEY ONLY --
55205 REM        --- OLD ROM ---
55210 WAIT516,1:GOSUB1000
55220 WAIT516,1,1:GOSUB2000:RETURN

55250 REM -- DETECT SHIFT KEY ONLY --
55255 REM        --- NEW ROM ---
55260 WAIT152,1:GOSUB1000
55270 WAIT152,1,1:GOSUB2000:RETURN
```

1.4 Branch on 'random' key.

I use this when the program waits for one of many possible single key responses. It saves line after line of IF.....THEN statements.

```
55300 REM  --- COMMAND PROMPT ---
55310 Z$="FGbNjXfPlKlnIZxO"
      :REM EXAMPLE RANDOM COMMANDS
55320 GETC$:IFC$=""GOTO55320
55330 FOR I=1 TO LEN(Z$)
55340 Q$=MID$(Z$,I1)
55350 IF Q$=C$ GOTO 55370
55360 NEXT: GOTO55320: REM INVALID
      RESPONSE
55370 ON I GOTO 100,200,300,400,500
      ,600,700,800,900,1000
55380 I=I-10: ON I GOTO 1100,1200
      ,1300,1400,1500,1600
```

1.5 Centre up text.

For a printer opened as file 1 with a line length of X and no TAB facility.

```
50500 REM --- CENTRE UP TEXT ---
50510 FOR Z=(X-LEN(Z$))/2 TO 1 STEP-1
      :IF Z<1 GOTO 55030
55020 PRINT#1," ";: NEXT
55030 PRINT#1,Z$: RETURN
```

INVERSE TRIGONOMETRIC FUNCTIONS

Here are a couple of handy methods of obtaining arc sine and arc cosine (remember, the result will be in radians).

```
10 DEF FNS(X)=ATN(X/SQR(1-X↑2))
20 DEF FNC(X)=ATN(SQR(1-X↑2)/X)
```

For those of you who are used to working in degrees, here are some handy user defined functions:

```
10 DEF FNS(X)=SIN(X/180/π)
20 DEF FNS(X)=SIN(X/180/π)
30 DEF FNS(X)=SIN(X/180/π)
```

These are three user defined function which when called with arguments an degrees will give the appropriate results In these examples V can be any variable bu if all three are defined in the sam programme, you must use three differen dummy variables.

EXAMPLE: PRINT FNS(30)

Result of this will be .5. Notice that th argument for FNS, or FN anything for tha matter, can be either a variable or numeri constant. Also, after a programm containing these definitions has been run these functions may be called using FN i the direct mode, that is, from the keyboar directly without being in a programme.

Mr. J. Smith of 38 Claremont Crescent, Croxley Green, Rickmansworth, Herts, WO 3QR
wrote in :

The error in the definition of arc cos should, I feel, be corrected. A possibl version is:- *

```
10 DEF FNC(X)=ATN(SQR(1-X↑2)/X)+(1-SGN(
   X))*π/2
```

This correctly gives (unless X=0) ar cos(-0.5) as 2.0943 (120 deg.); You formula gives arc cos(-0.5) as -60 deg. This would be incorrect in for example. "cosine rule" problem.

As you expect PET to be used in educationa establishments for solving trig. problems, I think it important to put this right.

* Note: if X is negative

    1-SGN(X) = 2

and if X is positive then:-

    1-SGN(X) = 0

This ensures that a correct multiple of pi is added to the arctangent.

W. J. Herbert
Department of Animal Services
Ninewells Hospital
Dundee DD2

You may like to have the enclosed short program for the 'Newsletter'. It eliminates itself!

This is not a joke but can be used e.g. to eliminate instructions before using a text processor so as to free working space for strings. In that case line 7 would be a 'GETA$' loop. Two or more, programs i.e. lines 8 & 9 can be placed

in sequence so as to eliminate more lines.

The program works as follows:

Line 8 prints the figures 1 to 9 down the screen and puts 9 'returns' into the keyboard buffer (poked into locations 527, 528 etc.)

Line 9 puts an additional 'return' into the buffer (536 by now) so that the instruction 'RUN100' is covered. This gets the program going again after 'END'. Three more 'Cursor ups' are printed than the number of lines to be deleted. Nine lines in this case, so 12 'Cursor ups'. Then the number of items in the buffer are put into location 525, and they are fired off by 'END'.

```
PROGRAM NAME: NEW ROM ELIMINATION

    1 REM
    2 REM
    3 REM
    4 REM
    5 REM
    6 REM
    7 REM
    8 PRINT"■■■■":FORI=1TO10:PRINTI:POKE622
      +I,13:NEXT
    9 PRINT"■RUN100■":POKE158,10:END
    100 PRINT"■ELIMINATED!"
```

```
PROGRAM NAME: ELIMINATION

    1 REM
    2 REM
    3 REM
    4 REM
    5 REM
    6 REM
    7 REM
    8 PRINT"■■■■":FORI=1TO10:PRINTI:POKE526
      +I,13:NEXT
    9 PRINT"■RUN100■":POKE525,10:END
    100 PRINT"■ELIMINATED!"
```

SYMBOLIC BASIC 'ASSEMBLER'

Mike Gross-Niklaus

1. PROCESSING A 'LISTED' BASIC TEXT

In the last issue, I described a method of processing BASIC listed text on tape to produce a formatted listing. The technique can be extended to actually change the text contents, which can then be fed back into the memory as a proper but amended program using our old friend, the Templeton-Butterfield merge.

It is not a straightforward matter to change for example all occurences of the variable T to become variable V, because of the problems in distinguishing variable T from variable CT or even PRINT.

2. A PERSONAL COMPUTER LANGUAGE

A solution to the problem is to head each variable you type in with a special character such as ! which is not part of the normal BASIC syntax. The function of this header character is to say "here is the start of a variable".

This means that what you type in is not BASIC but will have to be converted using an 'assembly' program.

Taking matters further, one can type in explicit variable names headed by ! and ending with a space say, which BASIC cannot differentiate but your assembler program can. For example, one can use such variable names as !PAYNET , !PAYGROSS , and even !PAYDEDUCTIONS , which BASIC would either see as variable PA or complain because it finds the reserved word 'TI' or 'ON' hidden inside the name.

The way an assembler handles these pseudo variables is to create a table of true variable assignments, allocating real variable names A - ZZ as it meets each new pseudo-variable.

When the program has been assembled onto a tape or disk file, your assembler can then print out this table sorted into both pseudo and real variable order. Such a printout makes an excellent addition to the program documentation.

For example:-

Pseudo text:-

```
10 FOR !INDEX =!START TO!FINISH
20 PRINT!NAMES$ (!INDEX)
30 NEXT
```

becomes:-

```
10 FORA=BTOC
20 PRINTA$(A)
30 NEXT
```

and the printed tables are:-

```
A=INDEX          FINISH=C
A$=NAMES$        INDEX=A
B=START          NAMES$=A$.
C=FINISH         START=B
```

3. JUMP TO LABELS

Taking the idea yet one more step, one

can overcome the problem of having to type in GOTO, GOSUB and THEN without yet knowing what line number you are going to jump to.

The idea is to use label references and labels again headed by special characters, perhaps and '.

Again a table is formed of all labels and their line numbers. A second pass picks up label references and converts them to line numbers using the table.

For example:-

Your text:-

```
10  START
20 INPUT"NUMBER 10 OR LESS";!NUMBER
30 IF!NUMBER>10THEN'ERROR
40 ?"YOUR NUMBER WAS 'OK'":END
50  ERROR
60 ?"YOUR NUMBER WAS TOO HIGH
70 GOTO'START
```

would assemble as:-

```
10 REM START
20 INPUT"NUMBER 10 OR LESS";A
30 IFA>10THEN50
40 ?"YOUR NUMBER WAS 'OK':END
50 REM ERROR
60 ?"YOUR NUMBER WAS TOO HIGH"
70 GOTO10
```

And once again your assembler would produce reference lists:-

```
ERROR 50        10 START
START 10        50 ERROR
```

Be warned that such assembly using tape and GET# is not fast and that your keyed in programs cannot be tested until they are assembled. However you will get considerable pleasure and satisfaction creating your own personal programming language and with large programs the technique may have some practical value.

"WHERE'D THE PENNY GO?"

Jim Butterfield

PET is certainly the greatest business tool since electric pencil sharpeners, and printers and floppy disks will herald an explosion of commercial applications.

Basic seems like the ideal language for a small business system - but it has a hidden "gotcha" that will give you problems if you don't know how to handle it. I call it, "the missing pennies problem", and it's common to almost all Basic implementations.

Crank up your PET and try this: PRINT 2.23 - 2.18 - - it's a simple business calculation and the answer has gotta be a nickel, right? So how come PET says .0499999998?

Think of the mess this could cause if you're printing out neat columns of pounds and pence results. Think of the problems if you arrange to print the first two places behind the decimal point: you'll print .04 instead of .05! Think of what the auditor will say when he finds that the totals don't add up correctly!

In a moment we'll discuss how to get rid of this problem. First, though, let's see how it happens.

PET holds numbers in floating binary. That means certain fractions don't work out evenly. Just as, in decimal, one third works out to .333333..., an endless number, PET sees fractions like .10 or .68 as endless repeating fractions - in binary. To fit the fractions in memory, it must trim it. Thus, many fractions such as .37 are adjusted slightly before storage.

Try this program: it will tell you how numbers are stored inside PET:

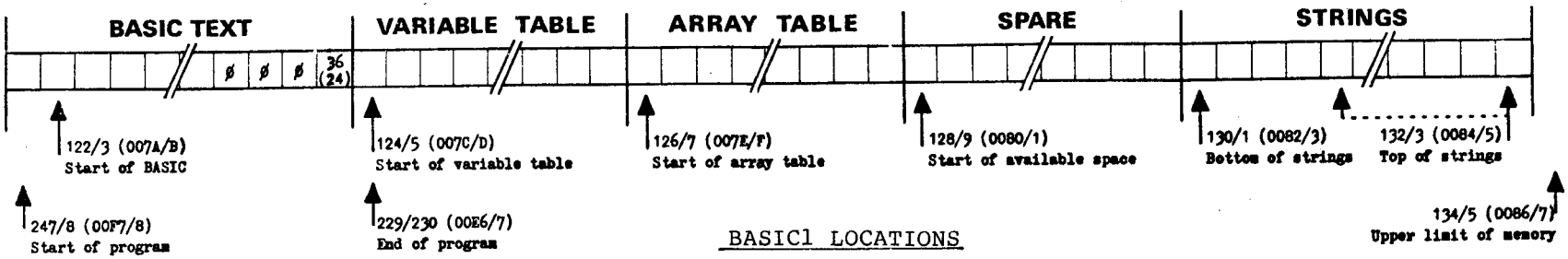
```
100 INPUT"AMOUNT";A: B=INT(A)
110 C=A-B: PRINTA"="B" ";
120 FOR J=1 TO 10: C=C*10: D=INT(C)
130 C=C-D: PRINTD;: IF C>0 THEN NEXTJ
140 PRINT: GOTO100
```

If you try entering numbers in our above example, 2.23 and 2.18, you'll see how PET stores them - and why the problems happen.

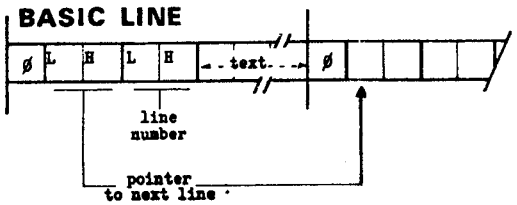
How to fix the problem. Easy. Change all numbers to pennies - which eliminates fractions - and your troubles disappear. For example:

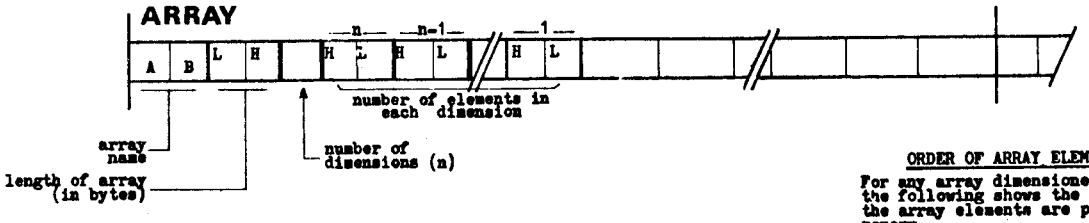
```
340 INPUT"AMOUNT";A
350 A=INT(A*100+.5)
360 PRINT"THE AMOUNT IS"A/100
```

Mr. R. Todd has sent in a very helpful chart which summarises the PET's use of memory for programs and variables - in a quick reference graphical form, and is printed overleaf.



BASIC1 LOCATIONS

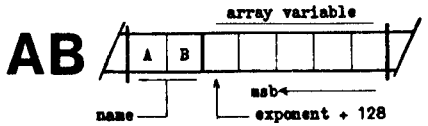


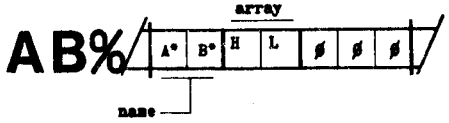


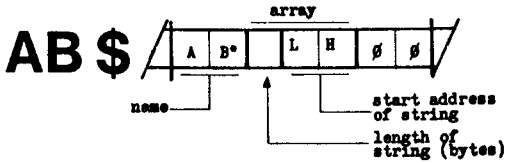
ORDER OF ARRAY ELEMENTS

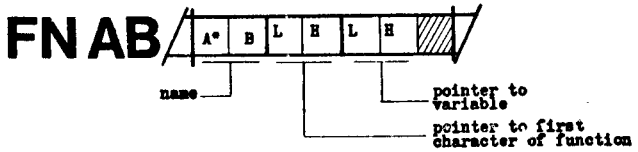
For any array dimensioned to (2,2,2) the following shows the order in which the array elements are placed in memory.

0, 0, 0
1, 0, 0
2, 0, 0
0, 1, 0
1, 1, 0
2, 1, 0
0, 2, 0
1, 2, 0
2, 2, 0
0, 0, 1
1, 0, 1
and so on...
to 2, 2, 2









PET REFERENCE SHEET 7: Memory allocation for BASIC program and variables

NOTES: All values are in decimal - those in brackets are in HEX
* indicates that bit 7 is set to 1
The array name takes the same form as the name for the ordinary variable
Sections of variables indicated as 'array' are those sections which are included in array tables

DYNAMIC LOADING

Steve Punter of Mississagua has a note for those performing LOADs from within programs. If strings are defined in text and are to be passed between programs they must be placed in high memory before the LOAD is executed.

As metioned earlier, a string variable is set up with only the length and a pointer to the location of the first character of the string. When a string is defined within BASIC text it is not moved into high memory, the pointer is set to look within the BASIC text. A dynamic LOAD replaces the text with the new text and although the variables remain intact, the string itself is lost. In other words, the pointer does not change but the information stored in memory does. So it is now likely that the string will contain BASIC commands and keywords.

The easiest way to avoid this is to define strings in text by concatenation. For example:-

```
10 SP$=""+"        "
20 NO$=""+"0123456790"
```

When a concatenation of any kind is performed, PET automatically rebuilds the string into high RAM thus protexting then from dynamic LOADs.

Finally, I have adapted a routine first sent in by Dr. I.C Smith of Queen Elizabeth College, London, for plotting the distribution of a variable. In his application, the variable (X) was read from a device on the 8-bit User port although it may be generated inside a program. Entered as it is printed here, the routine is being used to test (visually) the RND function on the PET.

PROGRAM NAME: DISTRIBUTION

```
1 DIM D(30): LS=33574
2 PRINT"□"
10 X=INT(RND(0.5)*30+1)
20 D(X)=D(X)+1
30 LC=INT(D(X)-8*INT(D(X)/8))
40 L=INT((D(X)-1)/8)
50 IFLC=0THENDC=160
60 IFLC=1THENDC=100
70 IFLC=2THENDC=111
75 IFLC=3THENDC=121
80 IFLC=4THENDC=98
90 IFLC=5THENDC=248
100 IFLC=6THENDC=247
110 IFLC=7THENDC=227
130 POKELS-40*L+X,DC
140 GOTO10
```

Notes:

1. The IF...THEN statements could be removed by means of a dimensioned variable holding the DC constants but are shown here to illustrate the operation of the routine.

2. LS is the location of the bottom left hand corner of the graph. It has been set here at 33574 to allow room for axis and labelling.

3. Any function could be put in place of line 10.

INPUT: SHORT FORM

D. J. POCOCK
Training Department

When writing BASIC programs with many different INPUT's many of you will have cursed. Because unlike most of the PET key words (PRINT etc) which have short forms of 2 or 3 characters (eg ? for PRINT) INPUT has no short form and all 5 characters have to be typed each time.

If you use '!' in place of INPUT when typing your program (eg 10 ! "WHAT IS YOUR NAME ";N$), it is possible with a small (approx 100 Byte) machine code routine to convert every occurance of '!' to INPUT.

The PET stores its key words as TOKENS in a single byte. For example INPUT is represented by 133(decimal) or $85(hex). It is this method of storage which makes this routine simple.

To be more specific the routine searches the BASIC text area for exclamation marks which are not enclosed in quotes, and converts these into the single character token for INPUT.

The routine can easily be altered to use other symbols (@ # & etc) to represent other basic key words. To alter the routine :-

POKE 943 with the ASC of the symbol

and POKE 944 with the value of the token (see THE PET REVEALED for a list of these).

The routine sits in the second cassette buffer and is called by SYS 826. Below are two methods for loading the routine, source code and a BASIC loader.

```
SOURCE*......PAGE 0001

LINE# LOC  CODE LINE


0001  0000             PTR     = $00
0002  0000             *=$033A
0003  033A  A9 00       BEGIN   LDA #$00
0004  033C  85 00               STA PTR         ;LO
0005  033E  A9 04               LDA #$04
0006  0340  85 01               STA PTR+1       ;HI BYTE START OF SEARCH
0007  0342  A0 00               LDY #$00        ;ZERO INDIRECT POINTER
0008  0344  20 99 03    ENDL    JSR INCR        ;
0009  0347  B1 00               LDA (PTR),Y     ;EXAMINE LINKS
0010  0349  8D AC 03            STA EP          ;FOR END OF
0011  034C  20 99 03            JSR INCR        ;PROGRAM
0012  034F  B1 00               LDA (PTR),Y     ;
0013  0351  0D AC 03            ORA EP          ;
0014  0354  F0 37               BEQ ENDP        ;TOTAL LINK = 0
0015  0356  20 99 03            JSR INCR        ;LINE # LO
0016  0359  20 99 03            JSR INCR        ;LINE # HI
3017  035C  A9 00               LDA #$00
0018  035E  8D AD 03            STA QTF         ;CLEAR QUOTE FLAG
0019  0361  8D AE 03            STA REMF        ;CLEAR REM FLAG
0020  0364  20 99 03    NCR     JSR INCR        ;FIRST / NEXT CHAR
0021  0367  B1 00               LDA (PTR),Y
0022  0369  F0 D9               BEQ ENDL        ;END OF BASIC LINE
0023  036B  C9 22               CMP #'"         ;QUOTES ?
0024  036D  F0 1F               BEQ QTMF        ;YES FLIP QUOTE FLAG
0025  036F  C9 8F               CMP #$8F        ;IS IT REM
0026  0371  D0 03               BNE CHAR        ;NO
0027  0373  8D AE 03            STA REMF        ;YES SET REM FLAG
0028  0376  CD AF 03    CHAR    CMP SYMBL       ;IS IT SYMBOL
0029  0379  D0 E9               BNE NCR         ;NO NEXT CHARACTER
0030  037B  AD AD 03            LDA QTF         ;YES CHECK QUOTE FLAG
0031  037E  D0 E4               BNE NCR         ;QUOTE FLAG SET
0032  0380  AD AE 03            LDA REMF
0033  0383  D0 DF               BNE NCR
0034  0385  AD B0 03            LDA TOKEN       ;LOAD KEY TOKEN
0035  0388  91 00               STA (PTR),Y     ;STORE IN BASIC TEXT
0036  038A  4C 64 03            JMP NCR         ;NEXT CHARACTER
0037  038D  60       ENDP    RTS             ;FINISHED TO BASIC
0038  038E  AD AD 03    QTMF    LDA QTF         ;
0039  0391  49 01               EOR #$01        ;FLIP QUOTE FLAG
0040  0393  8D AD 03            STA QTF         ;
0041  0396  4C 64 03            JMP NCR
0042  0399  18       INCR    CLC             ;INCREMENT POINTER
0043  039A  A5 00               LDA PTR
0044  039C  69 01               ADC #$01
0045  039E  85 00               STA PTR
0046  03A0  A5 01               LDA PTR+1
0047  03A2  69 00               ADC #$00
0048  03A4  85 01               STA PTR+1
0049  03A6  C9 80               CMP #$80
0050  03A8  F0 01               BEQ ERROR
0051  03AA  60                 RTS
0052  03AB  00       ERROR   BRK
0053  03AC  00       EP      .BYTE 0
0054  03AD  00       QTF     .BYTE 0
0055  03AE  00       REMF    .BYTE 0
0056  03AF  21       SYMBL   .BYTE '!'
0057  03B0  85       TOKEN   .BYTE $85
0058  03B1                   .END
```

PROGRAM NAME: INPUT USING !

```
10 REM ! TO GIVE INPUT
20 FOR L = 826 TO 944
30 READ C : POKE L , C
40 NEXT L
50 NEW
100 DATA169,0,133,0,169,4,133,1,160,0
110 DATA32,153,3,177,0,141,172,3,32,153
120 DATA3,177,0,13,172,3,240,55,32,153
130 DATA3,32,153,3,169,0,141,173,3,141
140 DATA174,3,32,153,3,177,0,240,217
150 DATA201,34,240,31,201,143,208,3,141
160 DATA174,3,205,175,3,208,233,173
170 DATA173,3,208,228,173,174,3,208,223
180 DATA173,176,3,145,0,76,100,3,96,173
190 DATA173,3,73,1,141,173,3,76,100,3
200 DATA24,165,0,105,1,133,0,165,1,105
210 DATA0,133,1,201,128,240,1,96,0,0,0
220 DATA0,33,133
```

PROGRAM NAME: TEST/DEMO FOR INPUT!

```
10 REM ** DEMONSTRATION / TEST PROGRAM
   **
20 SYS 826 : REM ENSURE ALL !'S ARE INP
   UT FOR THIS RUN
30 REM THIS IS A TEST !!!
40 ! "■■■WHAT IS YOUR NAME ";N$
50 ! "HOW OLD ARE YOU ";A
60 PRINT N$;" YOU ARE ABOUT";A*365;"DAY
   S OLD !!!"
70 GOTO 30
```

CHARACTER CODES

	OFF	RVS	CHR$	HEX
@	64	192	192	0
	0	128	64	
A	65	193	193	1
	1	129	65	
B	66	194	194	2
	2	130	66	
C	67	195	195	3
	3	131	67	
D	68	196	196	4
	4	132	68	
E	69	197	197	5
	5	133	69	
F	70	198	198	6
	6	134	70	
G	71	199	199	7
	7	135	71	
H	72	200	200	8
	8	136	72	
I	73	201	201	9
	9	137	73	
J	74	202	202	A
	10	138	74	
K	75	203	203	B
	11	139	75	
L	76	204	204	C
	12	140	76	
M	77	205	205	D
	13	141	77	
N	78	206	206	E
	14	142	78	
O	79	207	207	F
	15	143	79	

	OFF	RVS	CHR$	HEX
P	80	208	208	10
	16	144	80	
Q	81	209	209	11
	17	145	81	
R	82	210	210	12
	18	146	82	
S	83	211	211	13
	19	147	83	
T	84	212	212	14
	20	148	84	
U	85	213	213	15
	21	149	85	
V	86	214	214	16
	22	150	86	
W	87	215	215	17
	23	151	87	
X	88	216	216	18
	24	152	88	
Y	89	217	217	19
	25	153	89	
Z	90	218	218	1A
	26	154	90	
[	91	219	219	1B
	27	155	91	
\	92	220	220	1C
	28	156	92	
]	93	221	221	1D
	29	157	93	
↑	94	222	222	1E
	30	158	94	
←	95	223	223	1F
	31	159	95	

	OFF	RVS	CHR$	HEX
SP	96	224	160	20
	32	160	32	
!	97	225	161	21
	33	161	33	
"	98	226	162	22
	34	162	34	
#	99	227	163	23
	35	163	35	
$	100	228	164	24
	36	164	36	
%	101	229	165	25
	37	165	37	
&	102	230	166	26
	38	166	38	
'	103	231	167	27
	39	167	39	
(	104	232	168	28
	40	168	40	
)	105	233	169	29
	41	169	41	
*	106	234	170	2A
	42	170	42	
+	107	235	171	2B
	43	171	43	
,	108	236	172	2C
	44	172	44	
-	109	237	173	2D
	45	173	45	
.	110	238	174	2E
	46	174	46	
/	111	239	175	2F
	47	175	47	

	OFF	RVS	CHR$	HEX
0	112	240	176	30
	48	176	48	
1	113	241	177	31
	49	177	49	
2	114	242	178	32
	50	178	50	
3	115	243	179	33
	51	179	51	
4	116	244	180	34
	52	180	52	
5	117	245	181	35
	53	181	53	
6	118	246	182	36
	54	182	54	
7	119	247	183	37
	55	183	55	
8	120	248	184	38
	56	184	56	
9	121	249	185	39
	57	185	57	
:	122	250	186	3A
	58	186	58	
;	123	251	187	3B
	59	187	59	
<	124	252	188	3C
	60	188	60	
=	125	253	189	3D
	61	189	61	
>	126	254	190	3E
	62	190	62	
?	127	255	191	3F
	63	191	63	

RETURN			141	D
			13	
CLR SCREEN HOME			147	13
			19	
OFF RVS ON			146	12
			18	
CURSOR ↓			145	11
			17	
CURSOR →			157	10
			29	
INST DEL			148	14
			20	
RUN STOP			131	33
			3	

	105	233	169
✓	122	250	186
	94	222	222
	95	223	223

MACHINE LANGUAGE CASE CONVERTER

S. Donald Rossland,

This machine language program will convert strings to the correct upper/lower case condition for printing on CBM 2022/23 printers with an original ROM PET. It is relocatable so will operate anywhere in memory. The routine given here puts it in the second cassette buffer, but changing the location given in line 10100 will place it wherever you wish.

There are several things which must be done in order for the routine to operate correctly. These are best demonstrated by the following program.

```
0 ML$="" : GOSUB 10000
10 POKE 59468, 14
20 ML$="azl23AZ"
30 PRINT ML$ : OPEN 4,4 : PRINT#4,ML$
40 SYS 826
50 PRINT#4,ML$ : CLOSE 4
60 PRINT ML$
70 LIST
10000 DATA 160,   2, 177, 124, 141, 251
10010 DATA   0, 200, 177, 124, 141, 252
10020 DATA   0, 200, 177, 124, 141, 253
10030 DATA   0, 172, 251,   0, 136, 177
10040 DATA 252, 201, 219, 176,  22, 201
10050 DATA 193, 144,   5,  56, 233, 128
10060 DATA 208,  11, 201,  65, 144,   9
10070 DATA 201,  91, 176,   5,  24, 105
10080 DATA 128, 145, 252, 192,   0, 208
10090 DATA 223,  96
10100 FOR A = 826 TO 881 : READ B
10110 POKE A, B : NEXT : RETURN
```

Note that line 20 is altered once the program is RUN. This is done by the SYS command in line 40.

Now alter line 20 to:

```
20 ML$ = ML$ + "azl23AZ"
```

and reRUN from line 0. This time line 20 has not been changed in the listing. Whenever a string is formed by concatenation, the new string is stored in a location different from the original strings i.e. up in high RAM. It is this new location that has been altered. The major advantage in working on a string stored away from the program listing is that you don't have to worry if the string has been previously altered.

Now change line 0 to:

```
0 A = 0 : GOSUB 10000
```

and reRUN from line 0. Two points to note are:

1. Make sure that the variable string to be printed is #1 in the variable table, and

2. form the string to be printed by concatenation.

ASSEMBLY LANGUAGE LISTING OF UPPER/LOWER CASE CONVERTER

```
MOVE VARIABLE POINTERS TO ZERO PAGE

  LDY 2       Set Y register offset.
  LDA 124,Y   Load A with byte from
              variable table pointed to
              by 124/125 + Y
  STA 251,0   and move to location 251.
              This byte is the
              character count.
  INY         Increment offset.
  LDA 124,Y
  STA 252,0
  INY         Shift start address of
              string to zero page.
  LDA 124,Y
  STA 253,0

ADJUST STRING

  LDY 251,0   Load Y with string
              character count from
              location 251
  DEY         Decrement Y offset.
              Y points to character to
              be altered next.

TEST FOR LOWER CASE

  LDA 252,Y   Load A with string byte
              pointed to by 252/253
              and offset by Y
  CMP 219     and compare to lower
              case 'z' and
  BCS 22      if greater than, skip to
              COMPARE Y.
  CMP 193     Compare to lower case
              'a' and
  BCC 5       if less than skip to TEST
              FOR UPPER CASE.

ADJUST LOWER CASE

  SEC         Set carry flag
  SBC 128     Subtract 128 from the
              string byte in A and
  BNE 11      always skip to STORE
              MODIFIED CHARACTER.

TEST FOR UPPER CASE

  CMP 91      Compare to upper case 'Z'
              and
  BCS 9       skip to COMPARE Y if
              greater than.
  CMP 65      Compare to upper case 'A'
              and
  BCC 5       skip to COMPARE Y if less
              than.

ADJUST UPPER CASE

  CLC         Clear carry flag.
  ADC 128     Add 128 to string byte.

STORE MODIFIED CHARACTER

  STA 128,Y   Store byte at location
              pointed to by 252/253
              and offset by Y.
```

```
COMPARE Y FOR STRING END

  CPY Ø      Compare Y to 'Ø' and

  BNE 223    skip to DEY in ADJUST
             STRING if string not
             finished.

  RTS        Otherwise, return to BASIC.
```

FOR/NEXT LOOP STRUCTURE

Jim Butterfield

Recent remarks on popular BASIC implementations indicate that difficulties may be encountered if the programmer jumps out of a FOR/NEXT loop.

This would be very serious if true. The programmer performing a table search would be required to continue scanning the table even after finding the item he wants or to use the questionable practices such as meddling with the loop varaibles while still within a loop.

Fortunately, it is true only for a few complex situations and these are easy to fix if you understand how a dynamic FOR/NEXT loop works. Dynamic loops are those which are actualy set up as the program runs as opposed to pre-compiled loops which are checked out before the program starts to run.

When a dynamic interpreter such as Microsoft BASIC encounters a statement such as FOR J=.... it sets up internal tables to manage the loop. These internal tables contain such things as: where to return if a NEXT J is encountered, the identity of the loop variable ,in this case J, whether the loop is counting up or down etc.

These tables will remain until one of three things happen. If the loop goes through its complete range by encountering a suitable number of NEXT J statements, if a new FOR J... statement is found or if a higher priority loop is terminated for any of the above reasons.

The last rule is very sensible and it is worth a closer look. Suppose we have set up a sequence of commands such as: FOR I=... : FOR J=... : FOR K= ... and suppose the computer while dealing with these three loops finds a new FOR I=... statement. It very wisely says, in its own computerese, "OK - looks like the big loop is being restarted so the little ones are finished too". It promptly terminates the J and K loops, removing the tables from memory.

Exactly what we want - but there are a couple of hidden 'gotchas' that the user must know about when he gets into tricky coding routines.

The easiest one to spot is the situation where every loop has a different variable name. The first loop is, say, FOR A... the next one, FOR B... and the programmer continues through the alphabet with each new loop. His idea is good, he can analyse how each loop has behaved because each variable remains untouched for his examination. But each time he jumps out of a loop, the loop tables remain in memory, using up valuable stack or table space. He would be much better off to give at least his outer loops the same variable name and hence reclaim the space.

The second problem spot is a little more subtle and an example program would best illustrate this.

Here is a simple program to input a string, extract the individual words, eliminating multiple spaces and then printing them.

```
100 INPUT S$
110 K=0
120 FOR J=K+1 TO LEN(S$)
130 IF MID$(S$,J,1)<>" " GOTO 150
140 NEXT J
150 IF J>LEN(S$) GOTO900
160 FOR K=J TO LEN(S$)
170 IF MID$(S$,K,1)=" " GOTO 200
180 NEXT K
200 PRINT MID$(S$,J,K-J)
800 IF K<=LEN(S$) GOTO120
900 END
```

This program works quite well and is not too hard to follow. It should be noted that if either the J or K loops run to completion the variables will have a value of LEN(S$)+1; this is intended and is allowed for in lines 150 and 800.

Before we extend this program into catastrophe let us note one thing, by the time the program reaches line 200 both the J and K loops will still be open most of the time, we 'jumped out' of both of them. No real problem when we go back to 120, the new FOR J... will cancel them both.

Now to get the program into trouble. We may be writing a little ELIZA here and want to check the word we have found against a table of keywords so as to pick out a suitable reply. We will assume a table of twenty keywords and start to build a search loop. Replacing line 200 we start a new loop.

```
200 X$=MID$(S$,J,J-K)
210 FOR I=1 TO 20
```

Our loop is now three deep, J and K are still considered to be active, remember? No problem with three level loops; we are still OK.

But here is where we might get clever and wreck everything. We need to preserve K, that is where the search for the next word will occur but J has served its purpose and could be used again, right? Well... let us see.

This table of 20 words is really a double table. It contains pairs of words such as "I", "YOU", or "MY", "YOUR". To make our computer talk we must spot a word from either column, and switch in the word from the opposite column (so that "I HAVE FLEAS" will become "YOU HAVE FLEAS"). So we need one more loop to search over the two columns.

Let's be clever and use J, since we have decided that it isn't needed any more at this point. We code:

```
100 INPUT S$
110 K=0
120 FOR J=K+1 TO LEN(S$)
130 IF MID$(S$,J,I)<>" " GOTO 150
140 NEXT J
150 IF J>LEN(S$) GOTO900
160 FOR K=J TO LEN(S$)
170 IF MID$(S$,K,1)=" " GOTO 200
180 NEXT K
200 X$=MID$(S$,J,K-J)
210 FOR I=1 TO 20
220 FOR J=1 TO 2
230 IF X$=T$(I,J) THEN X$=T$(I,3-J):
    GOTO400
240 NEXT J
250 NEXT I
400 PRINTX$;" ";
800 IF K<=LEN(S$) GOTO120
900 END
```

Suddenly everything stops working and the world tumbles down around our program. What happened?

Let's stop and analyse. Just before executing line 220, the computer had three active loops with variables J, K and I. Now it reaches line 220, and what does it see? A loop based on J, the "biggest" loop! So what does it do? It cancels the K and I loops, of course, and starts a new J loop.

When we reach line 250, the computer sees NEXT I - but it no longer has an active FOR I= loop, and you get a NEXT WITHOUT FOR error message.

The rule here is slightly more complex, but not too tough. If you use J as an "outer" loop variable, never use if for an inner loop. If we reversed I and J in the coding from 210 to 250, we'd have no problem. Try to think in terms of the hierarchy of loops, and you can make sure that a given variable is used only at its proper hierarchal level.

Let's try to put the rules together and create a tiny ELIZA, polishing up some of the coding as we go. You'll have fun adding your own features to it.

```
100 DIM T$(1,4)
110 DATA ME,YOU,I,YOU,MY,YOUR,AM,ARE,MY
    SELF,YOURSELF
120 FOR J=0 TO 4
130 FOR K=0 TO 1
140 READ T$(K,J)
150 NEXT K: NEXT J
170 INPUT S$
180 K1=1:J1=1
190 FOR J=K1 TO LEN(S$)
200 IF MID$(S$,J,1)=" "THEN NEXTJ
210 J1=J
220 IF J>LEN(S$) GOTO900
230 FOR J=J1 TO LEN(S$)
240 IF MID$(S$,J,1)<>" "THEN NEXTJ
250 K1=J
260 X$=MID$(S$,J1,K1-J1)
270 FOR J=0 TO 4
280 FOR K=0 TO 1
290 IF T$(K,J)=X$ THEN X$=T$(1-K,J):
    GOTO320
300 NEXTK
310 NEXTJ
320 PRINT" ";X$;
330 IF K1<=LEN(S$) GOTO190
340 PRINT"?"
900 GOTO170
```

Note that the outermost loop is now always called J, the next down always K. I've tightened up the array to use the zero rows and columns to save memory; and the search loops are a little faster.

Even though the program is riddled with premature loop exits, there are no problems. Just observe a few simple rules and you will have efficient and trouble-free loops.

DISABLING THE STOP KEY

It's useful to be able to disable the STOP key, so that a program cannot be accidentally (or deliberately) stopped.

METHOD A is quick. Any cassette tape activity will reset the STOP key to its normal function however.

METHOD A, BASIC1:

Disable STOP key with POKE 537,136
Restore STOP key with POKE 537,133

METHOD A, BASIC2:

```
Disable STOP key with POKE 144,49
Restore STOP key with POKE 144,46
```

Method A also disconnects the computer's clocks (TI and TI$). If you need these for timing in your program, you should use method B.

METHOD B, BASIC1:

```
100 R$="20>:??:9??8=09024<88>6"
110 FORI=1TOLEN(R$)/2
120 POKEI+900,ASC(MID$(R$,I*2-1))
        *16+ASC(MID$(R$,I*2))-816
        NEXTI

    After the above has run:
Disable the STOP key with POKE 538,3
Restore the STOP key with POKE 538,230
```

METHOD B, BASIC2:

```
100 R$="20>:??:9??8=9;004<31>6"
110 FORI=1TOLEN(R$)/2
120 POKEI+844,ASC(MID$(R$,I*2-1))
        *16+ASC(MID$(R$,I*2))-816
        :NEXTI

    After the above has run:
Disable the STOP key with POKE 144,77
                         :POKE 145,3
Restore the STOP key with POKE 145,230
                         :POKE 144,46
```

How they work: Both methods change the interrupt program which takes care of the keyboard, cursor, clocks and the STOP key.

Method A simply skips the clock update and the STOP key test.

Method B builds a small program into the second cassette buffer which performs the clock update and STOP key test, but then nullifies the result of this test.

The little program in method B is contained in R$ in "pig hexadecimal" format. Machine language programmers would read this as:

```
20 EA FF (do clock update and STOP
         key test)
A9 FF 8D 9B 00 (cancel STOP test
               result)
4C 31 E6 (continue with keyboard
         service, etc.)
```

THE USE OF WAIT

The command WAIT can most usefully be used to test to see if a particular bit is set in a byte.

Implementing the command:-

The keyboard is buffered allowing up to 10 characters to be stored prior to processing. The number of keystrokes to be processed is stored in location 158 (525 Old ROM).

WAIT 158,1 will cause the current program to stop until a key is pressed. What happens is that current value of location 158 will be ANDed bit by bit with 00000001 until a match is made in one bit, then processing continues. This WAIT 158,1 will have the same effect as 10 GETA$:IFA$=""GOTO10.

To explain further here are two examples:

1. WAIT 158,8 (bit pattern 00001000). Each time a key is pressed location 158 will be incremented. No bit in 158 will match 00001000 until the key has been pressed eight times.

2. WAIT 158,7 (bit pattern 00000111) Will a key have to be pressed seven times before the programs continues? No. The first key pressed will have a bit pattern of 00000001 and this will make a bit match. Hence the program will continue.

A second argument may be added to the WAIT.
e.g. WAIT59410,4,4
This will WAIT until the space key has been pressed. If you take a PEEK at 59410 you will find that it contains 255 until a key on the same line as the STOP key is pressed (RVS 1 space < . =). The space key causes 251 (bit pattern 11111011) to appear in 59410.

It is not possible to use WAIT as any number other than 4 (0000100) will cause the WAIT to be fulfilled and 4 will cause the processor to wait indefinitely.

The first argument performs AND with location 59410.

```
          251 11111011
AND         4 00000100
              00000000
```

This result is then Exclusively ORed (EOR) with the second argument.

An Exclusive OR is the same as OR (01 OR 10 = 11) except that 1 only appears in the result when there is a 0 in the location and a 1 in the argument for the bit.

An Exclusive OR has the opposite effect to OR in that a bit is only set in the result when there is a zero in the location and a corresponding 1 in the argument. e.g.:

```
     10101010     11110000     11111111
EOR  00001111 EOR 00001111 EOR 10101010
     00000101     00001111     00000000
```

OR normally has this effect 10101010

```
                          OR  01010101
                              11111111
```

Going back to the result from WAIT 59410,4,4

```
     251   11111011
AND    4   00000100
           00000000
EOR    4   00000100
           00000100
```

And the program will continue.

BASIC REPEAT KEY

This little program gives you the chance of making a very simple repeat key for the PET.

Location 151 (515 old ROM) is used to hold the last key pressed. This location is the keyboard matrix number and while it can be used for fairly simple applications such as cursor control, it is not realistic to use it for ASCII conversion within a program.

The PET uses a ROM based look up table for converting the key matrix into ASCII. This table starts at 59127 (E6F7 hex). By adding on the key matrix number an ASCII character will be returned.

When the shift key is being pressed location 152 (516) goes from 0 to 1. When this is multiplied by 128 you will obviously get 0 or 128. By ORing this with the result from the PEEK at the lookup table the shifted ASCII characters are produced.

```
30 REM REPEAT - PAUL HIGGINBOTTOM
40 REM USE CURSOR CONTROLS TO MOVE *
80 PRINT"■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■";
90 A=(PEEK(59127+PEEK(151))OR(PEEK(152
   )*128)
100 IF A=253 GOTO90
110 IF A=17 THEN PRINT" ■■■■";
120 IF A=145 THEN PRINT" ■■■■";
130 IF A=157 THEN PRINT" ■■■■";
140 IF A=29 THEN PRINT" ■■■";
150 GOTO90
```

CHANGING ARRAY NAMES

Mike Gross-Niklaus

THE PROBLEM:-

Over the months, in CPUCN and other publications, many useful subroutines have been listed and explained. Some of these operate on the elements of an array.

But what if you want to use the same subroutine for several arrays? Most of us set up the subroutine with a 'working array', copy the contents of the array to be processed to the working array, perform the subroutine and then copy the processed working array back to the original one. Take a look at diagram 1 to see what I'm getting at.

The trouble is that if the array is a long one, this transferring can take a noticeable time. An alternative approach is to find the reference to the particular array in the array table, change the name to that of the 'working array', perform the subroutine, then change the name back again.

2. STEPS INVOLVED IN CHANGING THE NAME

First define the old and new names and convert them into the same form as held in the array table.

Then hunt through the table for the old name.

Finally change the name by poking the new code.

To change the array name back again involves the same process, reversing the order of the defined old and new names.

3. DEFINING THE NAMES

The technique shown here will only work on arrays of the same type (ie real, integer or string array.) So, assuming that you know what type of array the subroutine is going to process, only the alpha numeric parts of the array name need be specified. For example, whether you want to change XX$() to WK$() or XX() to WK(), the procedure will be the same.

The method used requires you to set up in a parameter string (A$), the old and new names as two characters each, using a space if the name is one character only. For example, to change XX$ to Z$ requires A$="XX,Z ".

This parameter string is then passed to a subroutine, 40900, which dissects it to produce the codes for each of the two bytes used by the BASIC interpreter for array names. The rules for these are set out in appendix A of the User Manual. Briefly, the ASCII codes of the alpha-numeric characters of the array name are used, with zero in the second byte if the name has only one character. To these values are added 128 in the second byte if the array is not a real number (floating point) array, and 128 to the first byte if it is an integer array. In the example shown, real arrays are assumed, so the only processing of the ASCII codes required is to replace CHR$(32), (space) by a zero.

4. HUNTING FOR THE CODED NAME

The interpreter maintains some useful pointers in the 'scratch pad' section of memory which you can get at with your BASIC programs by using PEEK.

Relevent here are the pointers for the start and end of arrays. The start is held as two bytes in locations 44 and 45, and the end in locations 46 and 47. To convert these references into decimal,

the second byte is multiplied by 256 and added to the first byte. For example if the first byte contains a PEEKed value of 4 and the second a value of 9, the decimal location is 9 times 256 plus 4 = 2308.

Using these calculations, you can determine the start and end of the search area.

The arrays are laid out with the first two bytes as the coded name and the third and forth defining how many bytes to the next array name. So the technique is to check the name pointed at by locations 44 and 45, then if there is no match, add the value of the next two bytes to the pointer and so on, until you find a match or you reach the end of the array area. This skipping search technique is exactly that employed by the interpreter every time it comes across a reference to an array element in your BASIC program, i.e. it searches from the start of the array table each time.

[Incidentally, the same technique is used by the interpreter for looking up normal variables and even line numbers during GOTOs and GOSUBS. Program operation speeds can be significantly improved by putting your most used variables, arrays and subroutines 'at the front of the queue!!']

5. EXAMPLE PROGRAM

The example program shows the use of a Bubble sort subroutine on two real number arrays XX and YY. Only 20 elements have been considered to allow listing of the unsorted and sorted arrays on the screen without scrolling or the need for screen page techniques.

The program fills the two arrays with random integer values in the range 1 to 99, and displays them for you to peruse. After you press the space bar, each array in turn has it's name changed to Z, the bubble sort is done on array Z, then the name is changed back again. Finally the sorted arrays are displayed with the oportunity for another run in case you can't beleive your eyes!

Incidently I lay out most of my programs in the manner shown if there is sufficient space. (qv this month's TEACHIN column.)

6. CONCLUSIONS

You may feel that it's a lot of work for a not very big reward! Well that depends on the size of your arrays and how frequently you process them during the run of the program. As with all subroutines, oncoG they are written and proven, they are available for instant use whenever you feel the need. In the case of one program I wrote to control industrial equipment, the increase in speed was essential. Using the normal transfer techniques would have caused the system to fail. The program was in PROM so I couldn't POKE the array name change into the BASIC text, which is messy anyhow if you are likely to amend the program with consequent changes in the relevent POKE locations. I could have reverted to machine code, but this method was far easier to implement, and fast enough.

I hope you get some mileage out of the idea, either by using it as intended, or just as an encouragement to you to have a look around the Scratch Pad (zero page) oocations and see what you can do by PEEKing and POKEing.

```
10 REM TITLE:- ARRAY NAME CHANGE
20 REM BY    :- MIKE GROSS NIKLAUS
30 REM FOR   :- CPCUN
40 REM DATE  :- 02/07/80
99 REM ==========
100 REM DEMONSTRATION ONLY
999 REM ==========
1000 REM PRELIMINARIES
1001 REM ----------
1010 DIM XX(20),YY(20): REM TWO ARRAYS
1020 A=RND (-TI): REM RANDOMISE
1030 PT=0: EA=0
1099 REM ==========
1100 REM FILL ARRAYS WITH RNDM NOS
1101 REM ----------
1110 DEF FNR (X)=INT (RND (1)*X+1)
1120 FOR I = 1 TO 20
1130 XX(I)=FNR(99): YY(I)=FNR(99)
1140 NEXT I
1199 REM ==========
1200 REM CONFIRM ORIGINAL SEQUENCE
1201 REM ----------
1210 GOSUB41000
1220 PRINT"■■PRESS SPACE BAR TO CONTINU
     E"
1230 Z$=" ":GOSUB50000
1299 REM ==========
2000 REM SORT ARRAY XX
2001 REM ----------
2010 A$="XX,Z ": GOSUB40000: IF EF = 1
     THEN END
2020 T$="XX": GOSUB50800
2030 A$="Z ,XX": GOSUB40000
2099 REM ==========
3000 REM SORT ARRAY YY
3001 REM ----------
3010 A$="YY,Z ": GOSUB40000: IF EF = 1
     THEN END
3020 T$="YY": GOSUB50800
3030 A$="Z ,YY": GOSUB40000
3099 REM ==========
4000 REM CONFIRM SORTED ARRAYS
4001 REM ----------
4010 GOSUB 41000
4099 REM ==========
5000 REM AGAIN?
5001 REM ----------
5010 PRINT"■■AGAIN (Y OR N)?
5020 Z$="YN":GOSUB50000:IF I=1 THEN
     RUN
5030 END
5099 REM ==========
40000 REM CODE OLD ARRAY NAME
40001 REM ----------
40010 X$=LEFT$ (A$,2): GOSUB 40900
40099 REM ==========
40100 REM FIND OLD ARRAY NAME
```

```
40101 REM ----------
40110 EF=0:REM ERROR FLAG RESET
40120 PT=PEEK(44)+PEEK(45)*256
40130 EA=PEEK(46)+PEEK(47)*256
40140 IF PEEK (PT)  <>NC(1) THEN 40160
40150 IF PEEK (PT+1)= NC(2) THEN 40200
40160 PT=PT+PEEK(PT+2)+PEEK(PT+3)*256
40170 IF PT<EA THEN 40140
40180 EF=1: REM ERROR FLAG
40190 RETURN
40199 REM ==========
40200 REM CHANGE THE NAME
40201 REM ----------
40210 X$= RIGHT$(A$,2):GOSUB 40900
40220 FORI = 0 TO 1 : POKE PT+I, NC(I+1
      ): NEXT I
40230 RETURN
40299 REM ==========
40900 REM CODE THE ARRAY NAME
40901 REM ----------
40910 FOR I = 1 TO 2: NC(I)=ASC (MID$ (
      X$,I,1)): NEXT I
40920 IF NC(2)=32 THEN NC(2)=0
40930 RETURN
40999 REM ==========
41000 REM CONFIRM ARRAYS
41001 REM ----------
41010 PRINT "SEQUENCE","ARRAY XX","ARR
      AY YY"
41020 FORI = 1 TO 20
41030 PRINT RIGHT$ (STR$ (I),2), XX(I),
      YY(I)
41040 NEXT I
41050 RETURN
41099 REM ==========
50000 REM AWAIT VALID KEY
50001 REM ----------
50010 GET A$: IF A$="" THEN 50010
50020 FOR I = 1 TO LEN (Z$)
50030 IF A$=MID$ (Z$,I,1) THEN RETURN
50040 NEXT I: GOTO 50010
50800 REM BUBBLE SORT
50801 REM ----------
50810 PRINT"SORTING ARRAY ";T$
50820 X=19: REM X  IS CURRENT BOTTOM
50830 SI=0: REM SI IS SWOP INDICATOR
50840 PRINT"SORTING COUNTDOWN";X;"
      "    :FOR I = 1 TO X
50850 IF Z(I)<=Z(I+1)THEN 50870
50860 A=Z(I): Z(I)=Z(I+1): Z(I+1)=A:SI
      =1
50870 NEXT I
50880 IF SI=1 THEN X=X-1: GOTO 50830
50890 RETURN
50899 REM ==========
59999 STOP
60000 REM SPUD
60001 REM ----------
60010 F$="1:"+LEFT$(TI$,4)
60020 FOR I = 1039 TO 1050
60030 A=PEEK(I): IF A=0 THEN 60060
60040 F$=F$+CHR$(A)
60050 NEXT I
60060 PRINT "SAVING "F$
60070 SAVE F$,8: VERIFY F$,8
60099 REM ----------
```

RE-DIMESIONING ARRAYS

You all know what happens when you attempt to re-define an array that has already been defined. PET returns a ?REDIM'D ARRAY ERROR. But maybe you had a good reason to re-dimension. And now you must perform a CLR which clobbers all your variables, or else work around it. No longer! By manipulating some pointers down in zero page, arrays can be REDIM'D with no problem at all. Try the following example:-

```
10 REM ---NEW ROM VERSION---
100 DIMA$(1000)
110 GOSUB2000
120 DIMA$(2000)
130 GOSUB2000
140 DIMA$(120)
150 END
2000 POKE46,PEEK(44)
2010 POKE47,PEEK(45)
2020 Z9=FRE(0):RETURN
```

```
10 REM ---OLD ROM VERSION---
100 DIMA$(100)
110 GOSUB2000
120 DIMA$(200)
130 GOSUB2000
140 DIMA$(120)
150 END
2000 Z9=PEEK(126):POKE128,Z9
2010 Z9=PEEK(127):POKE129,Z9
2020 Z9=FRE(0):RETURN
```

The subroutine at 2000 "squeezes" the array out by making the End of Arrays Pointer equal to the Start of Arrays Pointer. PET now believes that there are no arrays of any name so DIMensioning is ok. The new array(s) is "built" in the same memory space.

Line 2000 forces a "garbage collection" so that any strings associated with Array A$ are thrown away. This wouldn't really be necessary with floating point or integer arrays since the values are stored in the array itself. With string arrays, only the string lengths and pointers to the strings are stored in the array. The strings lie elsewhere in RAM; in high memory if they were the result of a concatenation or INPUT from the keyboard, disk, etc. and directly in text if that is where they were defined (why store it twice?). This is also true for non-array type string variables. Of course, strings residing in text are not thrown away by garbage collection.

This trick can be played especially well when the sizes of your arrays are maintained in a disk file along with the file information.

Sometimes clearing all the arrays may not always be desirable. In that case, the order in which the arrays are defined becomes important. The 'permanent' arrays must be DIMensioned first, 'temporary' arrays last. However, if the value of the End of Arrays Pointer is stored immediately after defining the last 'permanent' array, the 'temporary' arrays can be squeezed out by POKing the

End of Arrays Pointer with this value later on. For example:

```
5 A=0
10 REM ---NEW ROM VERSION---
15 PL%=0:PH%=0
100 DIMA$(100),B(200),C(300)
120 POKE826,PEEK(46): POKE827,PEEK(47)
130 GOSUB2000
140 DIMA(A)
145 A=A+10:PRINTA:GOTO130
150 END
2000 POKE46,PEEK(826): POKE47,PEEK(827)

2020 RETURN
```

The subroutine at 2100 would allow Arrays X, Y% and Z$ to be redimensioned any number of times without destroying Arrays A, B% and C$.

RESTORE DATA LINE PROGRAM

Paul Barnes

DESERONTO, ONTARIO

Executing the RESTORE command causes the next READ to occur at the very first DATA element in your program. This subroutine can be used to RESTORE the DATA line pointer at a line other than the first.

It doesn't matter if you don't give the number of the line that has the "DATA" keyword in it that you want to start at, as long as it is past previous DATA statements so that the next data to be read will be the one desired.

```
4 REM*********************************
5 REM***  RESTORE DATA LINE PRG  ***
6 REM***      BY PAUL BARNES     ***
7 REM***   DESERONTO, ONTARIO    ***
8 REM***      NEW ROM VERSION    ***
9 REM*********************************
10 DATA 166,60,134,17,166,61
15 DATA 134,18,32,44,197,144
20 DATA 11,166,92,142,132,3
25 DATA 166,93,142,133,3,96
30 DATA 162,0,142,132,3,162
35 DATA 0,42,133,3,96
40 FOR F=826 TO 860: READS: POKEF,S:
   NEXT
50 DATA"GOOD-BYE!"
60 DATA"ANYBODY HOME?"
70 J=26545*10: FOR D=1 TO 100: DATA"MAY
   BE!"
80 NEXT: DATA"HI!"
100 DATA"GO HOME!"
110 DATA"GO DIRECTLY TO JAIL!"
120 DATA"DO NOT PASS GO!"
130 DATA"DO NOT COLLECT 200 POUNDS!!!"
200 GOSUB1000
210 FOR T=1 TO 3: READ A$: PRINTA$
230 NEXT: PRINT: GOTO200
998 REM***SUBROUTINE TO RESTORE DATA**
999 REM*** AT A CERTAIN LINE NUMBER ***
1000 INPUT"RESTORE TO LINE";A
1010 H=INT(A/256): L=A-H*256
1020 REM POKE CURRENT DATA LINE POINTER

1030 POKE60,L: POKE61,H
1040 SYS826
1050 L=PEEK(900):H=PEEK(901)
1060 IF L=0 AND H=0 THENPRINT"LINE NOT
     FOUND": GOTO1000
1070 REM POKE MEMORY ADDRESS OF DATA LI
     NE
1080 A=H*256+L-1: H=INT(A/256): L=A-H*2
     56
1090 POKE62,L: POKE63,H
1100 RETURN
```

```
4 REM*********************************
5 REM***  RESTORE DATA LINE PRG  ***
6 REM***      BY PAUL BARNES     ***
7 REM***   DESERONTO, ONTARIO    ***
8 REM***      OLD ROM VERSION    ***
9 REM*********************************
10 DATA 166,142,134,8,166,143
15 DATA 134,9,32,34,197,144
20 DATA 11,166,174,142,132,3
25 DATA 166,175,142,133,3,96
30 DATA 162,0,142,132,3,162
35 DATA 0,42,133,3,96
40 FOR F=826 TO 860: READS: POKEF,S:
   NEXT
100 REM==============================
110 REM=LINES 50 TO 230 ARE THE SAME=
120 REM==============================
998 REM***SUBROUTINE TO RESTORE DATA***

999 REM*** AT A CERTAIN LINE NUMBER ***

1000 INPUT"RESTORE TO LINE";A
1010 H=INT(A/256): L=A-H*256
1020 REM POKE CURRENT DATA LINE POINTER

1030 POKE142,L: POKE143,H
1040 SYS826
1050 L=PEEK(900):H=PEEK(901)
1060 IF L=0 AND H=0 THENPRINT"LINE NOT
     FOUND": GOTO1000
1070 REM POKE MEMORY ADDRESS OF DATA LI
     NE
1080 A=H*256+L-1: H=INT(A/256): L=A-H*2
     56
1090 POKE144,L: POKE145,H
1100 RETURN
```

TRACE

Brett Butler

Toronto

Yet another of the superb programs brought over the Atlantic by Jim Butterfield on the occasion of his last visit to Englannd. This Trace program by Brett Butler is a very useful tool for debugging BASIC programs. The whole of the line currently being executed is displayed at the top of the screen in a reverse field window. This is far better than many of the Trace routines available now which display only the line number.

The rate at which your program will execute during debugging can be fine tuned with a SYS command and another SYS command will switch the Trace off again. When the Trace is enabled your program can be run at approximatly one quarter of the normal speed by simply depressing the shift key.

This incredible machine code program works equally well with both BASIC1 and BASIC2 PETs and will locate itself into any memory size. It will also sit happily together with any other machine code utility which may be resident in memory such as SUPERMON, locating itself so that all the facilities are available concurrently.

The 'intelligence' built into Trace's locating routine can lead to some unexpected side effects. If you load and run the BASIC loader for Trace the program will give you the SYS locations appropriate to your particular PET. If you ignore the instruction to note these down somewhere you may need to load Trace again. If you do this a number of times you will notice the SYS locations are different each time. A quick memory check - ?FRE(0) - will show you that the available memory decreases every time the program is run. What is happening? The BASIC loader is locating Trace at the top of available memory. When it has finished loading the program is decreases the top of memory to the start of Trace thus protecting itself from BASIC. If you require the missing memory then switch off your PET and start again.

running. To STOP the program whilst in the delayed state, press RUN-STOP and SHIFT together.

There is a short-cut in the program which can cause the TRACE to miss the start of a new line. It occurs if a GOTO is performed from a line number 256*K+J (where J is between 0 and 255 and K is any integer) to line 256*K, e.g. line 260 to line 256, or line 1300 to line 1280.

Users will have noticed that, when TRACE displays a READ statement, the characters read appear in the displayed line immediately following the READ.

MODIFICATIONS TO TRACE TO TEST SHIFT CONTINUOUSLY

Changes are underlined.

```
104 DATA 253,208,4,228,254,240,106,133,
    253,133,35,134,254,134,36,169
120 DATA 3,133,107,165,152,208,10,202,
    208,249,136,208,246,198,107,208
136 DATA 242,32,-54,169,160,160,80,153,
    255,127,136,208,250,132,182,132
860 DATA 228,78,240,107,133,77,133,82,
    134,78,134,83,169,3,133,74,173,4
870 DATA 2,208,10,202,208,248,136,
    208,245,198,74,16,241,32,-54,
    169,160
1120 PRINT "CHANGE SPEED WITH: POKE";S1
     +119;",X"
```

NOTE: The modifications have been tested on an upgraded ROM machine (model3032) but not an original machine

TRACE

A. G. Price

Principal Lecturer Mathematics
Liverpool Polytechnic

"PET users may be interested in my experience with Brett Butler's TRACE routine, published in CPUCN Volume 2 Issue 3. As written, tracing can be slowed down by use of the POKE instruction indicated. This introduces a delay of about 300 milliseconds multiplied by the value POKEd, so that the built-in value of 3 causes execution at about 1 second per line. The delay for a value of 1 is rather variable, and is not recommended. A value of 0 corresponds to 256, and gives a delay of about 80 seconds. The delay is not interruptable by use of either the SHIFT or STOP keys, which can be a little frustrating.

By rearranging the TRACE program slightly, it can be made to respond to the SHIFT key at any time, thus giving the effect of a 'single-shot' key by setting a large delay and tapping the SHIFT key to advance to the next instruction. Hold it down for full-speed

```
 50 PRINT"■■■■THIS PROGRAM LOCATES ■TRAC
    E IN"
 60 PRINT"ANY SIZE MEMORY THAT IS FITTED
    ..."
 65 IFPEEK(65E3)=254THEND=2:E=52:GOTO70
 66 IFPEEK(65E3)<>192THENPRINT"?? I DON'
    T KNOW YOUR ROM ??":END
 67 D=1:E=134:FORJ=1TO1E3:READX:IFX<1E4
    THENNEXTJ
 70 PRINT"I SEE YOU HAVE AN ";
 71 IFD=1THENPRINT"ORIGINAL";
 72 IFD=2THENPRINT"UPGRADE";
 73 PRINT" R O M."
 98 DATA -342,162,5,189,249,224,149,112,
    202,16,248,169,239,133,128,96
 99 DATA 173,-342,133,52,173,-341,133,53
    ,169,255,133,42,160,0,162,3
100 DATA 134,43,162,3,32,-271,208,249,2
    02,208,248,32,-271,32,-271,76
101 DATA 121,197,162,5,189,-6,149,112,2
    02,16,248,169,242,133,128,96
102 DATA 230,42,208,2,230,43,177,42,96,
    230,119,208,2,230,120,96
103 DATA 32,115,0,8,72,133,195,138,72,1
    52,72,166,55,165,54,197
104 DATA 253,208,4,228,254,240,106,133,
    253,133,35,134,254,134,36,165
120 DATA 152,208,14,169,3,133,107,202,2
    08,253,136,208,250,198,107,208
136 DATA 246,32,-54,169,160,160,80,153,
    255,127,136,208,250,132,182,132
```

```
153 DATA 37,132,38,132,39,120,248,160,1
    5,6,35,38,36,162,253,181
169 DATA 40,117,40,149,40,232,48,247,13
    6,16,238,216,88,162,2,169
185 DATA 48,133,103,134,102,181,37,72,7
    4,74,74,74,32,-44,104,41
202 DATA 15,32,-44,166,102,202,16,233,3
    2,-38,32,-38,165,184,197,119
221 DATA 240,55,165,195,208,4,133,253,2
    40,47,16,42,201,255,208,8
237 DATA 169,105,32,-30,24,144,33,41,12
    7,170,160,0,185,145,192,48
254 DATA 3,200,208,248,200,202,16,244,1
    85,145,192,48,6,32,-32,200
271 DATA 208,245,41,127,32,-32,165,119,
    133,184,104,168,104,170,104,40
288 DATA 96,168,173,64,232,41,32,208,24
    9,152,96,9,48,197,103,208
304 DATA 4,169,32,208,2,198,103,41,63,9
    ,128,132,106,32,-54,164,182
322 DATA 153,0,128,192,195,208,2,160,7,
    200,132,182,164,106,96,76
333 DATA -255,32,-262
700 DATA 1E10
800 DATA-343,162,5,189,181,224,149,194,
    202,16,248,169,239,133,210,96
810 DATA173,-343,133,134,173,-342,133,1
    35,169,255,133,124,160,0,162
820 DATA3,134,125,162,3,32,-272,208,249
    ,202,208,248,32,-272,32,-272
830 DATA76,106,197,162,5,189,-6,149,194
    ,202,16,248,169,242,133,210,96
840 DATA230,124,208,2,230,125,177,124,9
    6,230,201,208,2,230,202,96,32
850 DATA197,0,8,72,133,79,138,72,152,72
    ,166,137,165,136,197,77,208,4
860 DATA228,78,240,107,133,77,133,82,13
    4,78,134,83,173,4,2,208,14,169
870 DATA3,133,74,202,208,253,136,208,25
    0,198,74,16,246,32,-54,169,160
880 DATA160,80,153,255,127,136,208,250,
    132,76,132,84,132,85,132,86,120
890 DATA248,160,15,6,82,38,83,162,253,1
    81,87,117,87,149,87,232,48,247
900 DATA136,16,238,216,88,162,2,169,48,
    133,89,134,88,181,84,72,74,74
910 DATA74,74,32,-44,104,41,15,32,-44,1
    66,88,202,16,233,32,-38,32,-38
920 DATA165,75,197,201,240,55,165,79,20
    8,4,133,77,240,47,16,42,201,255
930 DATA208,8,169,94,32,-30,24,144,33,4
    1,127,170,160,0,185,145,192,48
940 DATA3,200,208,248,200,202,16,244,18
    5,145,192,48,6,32,-32,200,208
950 DATA245,41,127,32,-32,165,201,133,7
    5,104,168,104,170,104,40,96,168
960 DATA173,64,232,41,32,208,249,152,96
    ,9,48,197,89,208,4,169,32,208
970 DATA2,198,89,41,63,9,128,132,81,32,
    -54,164,76,153,0,128,192,79,208
980 DATA2,160,7,200,132,76,164,81,96,76
    ,-256,32,-263
1000 S2=PEEK(E)+PEEK(E+1)*256:S1=S2+D-3
     44
1010 FORJ=S1TOS2-1
1020 READX:IFX>=0GOTO1050
1030 Y=X+S2:X=INT(Y/256):Z=Y-X*256
1040 POKEJ,Z:J=J+1
1050 POKEJ,X
1060 NEXTJ
1070 PRINT"■■ === TRACE ==="
1080 REMARK: BY BRETT BUTLER, TORONTO
1090 PRINT"■TO INITIALIZE AFTER LOAD: S
     YS";S1+17
1100 PRINT"TO ENABLE TRACE:      SYS";S
     1+56
1110 PRINT"TO DISABLE:      SYS";S1+2
1120 PRINT"■CHANGE SPEED WITH:  POKE";S
     1+125-D;"■■,X"
1130 PRINT"■==MAKE A NOTE OF ABOVE COMM
     ANDS=="
1140 PRINT"■SAVE USING MACHINE LANGUAGE
      MONITOR:"
1150 PRINT"   .S ";
1160 S=INT(S1/256):T=S1-S*256
1170 POKEE,T:POKEE+1,S
1180 POKEE-4,T:POKEE-3,S
1190 IFD=2THENPRINTCHR$(34);"TRACE";
     CHR$(34);",01";
1195 IFD=1THENPRINT" 01,TRACE";
1200 S=S1:GOSUB1400
1210 S=S2:GOSUB1400
1220 PRINT:END
1400 PRINT",";:S=S/4096
1410 GOSUB1420
1420 GOSUB1430
1430 T=INT(S):IFT>9THENT=T+7
1440 PRINTCHR$(T+48);:S=(S-INT(S))*16:
     RETURN
```

CROSS—REFERENCE

Jim Butterfield

Toronto

One of the handy things about the 2040 disk system is that it allows you to read programs - or write them, for that matter - as if they were data files.

The possibilities are endless: you can analyze or cross-reference programs; renumber them; repack them into minimum number of lines deleting spaces, comments, etc.; or even create a program-writing program that is tailor-made for a particular job.

There are two types of cross-reference normally needed for a BASIC program. It's written in BASIC: that means that it won't run too fast (all those GET statements) but you can read what it's doing fairly easily.

There are two types of cross-references normally needed for a BASIC program. One is the variable cross-reference: where do I use B$? The other is a line-number cross-reference: when do I go to line 360 ? CROSS-REF does either. An example of both types is shown - the program in this case did the cross-references of itself.

READING A BASIC PROGRAM AS A FILE

To read a BASIC program, you must OPEN it as a file, using type P for PRG rather than S for SEQ. Line 170 of CROSS-REF does this.

If you read a zero character from the program (that's CHR$(0), not ASCII zero which has a binary value of 48), the GET# command gives you a small problem: it will give you a null string instead of the CHR$(0) you might normally expect. You need to watch this condition and

correct it where necessary: you'll see this type of coding in lines 260, 270 and 300.

The first thing to do when you OPEN the file is to get the first two bytes. These represent the program start address, and should be CHR$(1) and CHR$(4) for a normal BASIC program starting at hexadecimal 0401 (see line 180).

Now you're ready to start work on a line of BASIC. The first two bytes are the forward chain. If they are both zero (null string) we have reached the end of the BASIC program; otherwise, we don't need them for this job (see line 240).

Continuing on the BASIC line: the next pair of bytes represent the line number, coded in binary. We're likely to need this, so we calculate it as L (lines 260 and 280) and also create it's string equivalent, L$. We take an extra moment to right-justify the string by putting spaces at the front so that it will sort into proper numeric order.

From this point on we are looking at the text of the BASIC line until we reach a zero which flags end-of-line. At that time we go back and grab the next line.

DETAILED SYNTAX ANALYSIS

When digging out variables or line numbers, we have several jobs to do. As we look through the BASIC text, we must find out where the variable or line number starts. For a variable, that's an alphabetic character; for a line number, it's the preceeding keyword GOTO, GOSUB, THEN or RUN followed by an ASCII numeric.

Once we've "acquired" the variable or line number, we must pick up its following characters and tack them on. For line numbers it's strictly numeric digits. For variables, things are more complex. Both alphabetic and numeric digits are allowed, but we should throw away all after the first two since GRUMP and GROAN are the same variable (GR) in PET BASIC. We must also pick up a type identifier - % for integer variables or $ for strings - if present. Finally, we have to spot the left bracket that tells us we have an array variable.

To help us do this rather complex job, we construct a character type table. Each entry in the table represents an ASCII character, and classifies it according to its type. Numeric characters are type 6. If we're looking for variables, alphabetic characters are type 5, identifiers are type 7, and the left bracket is type 8.

To help us in scanning the BASIC line, we define the end-of-line character as type 0; the quotation mark as type 2; the REM token as type 3; and the DATA token as type 4.

Every time we get a new character from BASIC, we get its type from table C as variable C9. If we're looking for a new variable or line number, we see if it matches C - alphabetic for variables, numeric for line numbers. Once we find the new item, we kick C out of range and start searching based on the value of C1. This mechanism means that we can search for a variable starting with an alphabetic, and then allow the variable to continue with alphabetics, numerics or whatever.

To summarize variables in this area: A is the identity of the character we have obtained from the BASIC program, and C9 is its type. If we're searching, C is the type we are looking for; otherwise it's kicked out of range, to -1 or 9. C1 tells us we're collecting characters and what type we're allowed to collect. C2 is our variables/line numbers flag; it tells us what we're looking for. M$ is the string we've assembled.

The routine from 480 to 520 scans ahead to skip over strings in quotes and DATA and REM statements.

COLLECTING THE RESULTS

For each line of the BASIC program we are analyzing, we collect and sort any items we find, eliminating duplicates. They are staged in array A$ in lines 320 to 370.

When we are ready to start a new line, we add this table to our main results table, array X$, in lines 200 to 220. To save sorting time, we merge these pre-sorted values into the main table. At this point, our data has the line number stuck on the end; this way, we're handling two values within a single array.

Because the merging of the two tables must start at the top so that we can make room for the new items, the items are handled in reverse alphabetic order. We print this to the screen so that you can watch things working. At BASIC speed, this program can take quite a while to run; it's nice to confirm that the computer is doing something during this period.

FINAL OUTPUT

We finish the job starting at line 530. It's mostly a question of breaking the stuck-together strings apart again and then checking to see if we need to start a new line.

DO YOUR OWN THING

The size of @ array X$ determines how large a program you can handle. The given value of 500 is about right for 16K machines; with 32K you can raise it to 1500 or so.

If you're squeezed for space, change array C to an integer array C%. As you

can see from the cross reference listing, you'll need to change lines 100, 140, 150, 160 and 310 - see how handy the program is ?

As mentioned before, run time is slow. A machine language version - or even a BASIC program with machine language inserts - would speed things up dramatically.

NOTE: Some ASCII printers may give double spaced output. If this is a problem the PRINT#2 statements in 590 and 610 should be changed to PRINT#2,CHR$(13);.

PROGRAM NAME: CROSS REFERENCE

```
 100 DIM A$(15),B$(3),X$(500),C(255)
 110 PRINT"CROSS-REF    JIM BUTTERFIELD"

 120 Q$=CHR$(34):S$="      ":B$(1)=Q$:B$(
     3)=CHR$(58)
 130 INPUT"VARIABLES OR LINES";Z$:C2=5:
     IFASC(Z$)=76THENC2=6
 140 FORJ=1TO255:C(J)=4:NEXTJ:FORJ=48TO5
     7:C(J)=6:NEXTJ
 150 IFC2=5THENFORJ=65TO90:C(J)=5:NEXTJ:
     FORJ=36TO38:C(J)=7:NEXTJ:C(40)=8
 160 C(34)=1:C(143)=2:C(131)=3
 170 INPUT"PROGRAM NAME";P$:OPEN1,8,3,"0
     :"+P$+",P,R"
 180 GET#1,A$,B$:IFASC(B$)<>4THENCLOSE1:
     STOP
 190 IFB=0GOTO240
 200 PRINTL$;:K=X:FORJ=BTO1STEP-1:PRINT"
      ";A$(J);:X$=A$(J)
 206 X$=X$+L$
 210 IFX$(K)>=X$THENX$(K+J)=X$(K):K=K-1:
     GOTO210
 220 X$(K+J)=X$:NEXTJ:X=X+B:PRINT:B=0
 230 REM  GET NEXT LINE, TEST END
 240 GET#1,A$,B$:IFLEN(A$)+LEN(B$)=0
     GOTO530
 250 REM GET LINE NUMBER
 260 GET#1,A$:L=LEN(A$):IFL=1THENL=ASC(A
     $)
 270 GET#1,A$:A=LEN(A$):IFA=1THENA=ASC(A
     $)
 280 C=C2:C1=-1:L=A*256+L:L$=STR$(L):IF
     LEN(L$)<6THENL$=LEFT$(S$,6-LEN(L$))
     +L$
 290 REM GET BASIC STUFF
 300 GET#1,A$:A=LEN(A$):IFA=1THENA=ASC(A
     $)
 310 C9=C(A):IFC9>C1GOTO380
 320 IFC2=6ANDLEN(M$)<5THENM$=" "+M$:
     GOTO320
 326 K=0:IFB=0GOTO360
 330 FORJ=1TOB:IFA$(J)=M$GOTO370
 340 IFA$(J)<M$THENNEXTJ:K=B:GOTO360
 350 FORK=BTOJSTEP-1:A$(K+1)=A$(K):NEXTK

 360 B=B+1:A$(K+1)=M$
 370 C=C2:C1=-1:M$=""
 380 IFC2=5GOTO420
 390 IFA=137ORA=138ORA=141ORA=167THENC=6
     :GOTO470
 400 IFA=44ORA=32GOTO470
 410 IFC9<>6THENC=9:GOTO470
 420 IFC9=CTHENC=-1:C1=4
 430 IFC>6GOTO470
 440 IFC<0ANDC9>C1ANDC9>6THENC1=C9:GOTO4
     60
 450 IFC2=5THENIFLEN(M$)>20RC>0GOTO470
 460 M$=M$+A$
 470 ONC9+1GOTO190,480,480,480:GOTO300
 480 B$=B$(C9):C$=""
 490 GET#1,A$:IFA$=""GOTO190
 500 IFA$=B$GOTO300
 510 IFA$<>0$GOTO490
 520 A$=B$:B$=C$:C$=A$:GOTO490
 530 CLOSE1:INPUT"PRINTER";Z$
 540 C=3:Z=6:IFASC(Z$)=89THENC=4:Z=12
 550 OPEN2,C:PRINT#2:PRINT#2,"CROSS REFE
     RENCE - PROGRAM ";P$
 560 X$="":FORJ=1TOX:A$=X$(J)
 565 IFC2=6THENK=6:GOTO580
 570 FORK=1TOLEN(A$):IFMID$(A$,K,1)<>" "
     THENNEXTK:STOP
 580 B$=LEFT$(A$,K-1):C$=MID$(A$,K+1):
     IFX$=B$GOTO600
 590 PRINT#2:Y=0:X$=B$:PRINT#2,X$;LEFT$(
     S$,5-LEN(X$));
 600 Y=Y+1:IFY<ZGOTO620
 610 Y=1:PRINT#2:PRINT#2,S$;
 620 PRINT#2,LEFT$(S$,6-LEN(C$));C$;
 630 NEXTJ:PRINT#2:CLOSE2
```

```
CROSS REFERENCE - PROGRAM CROSS-REF

(       270   280   300   310   390   400
A$      190   240   260   270   300   460   490   500   510   520   560
        570   580
A$(     100   200   330   340   350   360
B       190   200   220   326   330   340   350   360
B$      190   240   480   500   520   580   590
B$(     100   120   480
C       280   370   390   410   420   430   440   450   540   550
C$      480   520   580   620
C(      100   140   150   160   210
C1      280   310   370   420   440
C2      130   150   280   320   370   380   450   565
C9      310   410   420   440   470   480
J       140   150   200   210   220   330   340   350   560   630
K       200   210   220   326   340   350   360   565   570   580
L       260   280
L$      200   206   280
M$      320   330   340   360   370   450   460
P$      170   550
Q$      120   510
S$      120   280   590   610   620
X       200   220   560
X$      200   206   210   220   560   580   590
X$(     100   210   220   560
Y       590   600   610
Z       540   600
Z$      130   530   540
```

```
CROSS REFERENCE - PROGRAM CROSS-REF

  190   470   490
  210   210
  240   190
  300   470   500
  320   320
  360   326   340
  370   330
  380   310
  420   380
  460   440
  470   390   400   410   430   450
  480   470
  490   510   520
  530   240
  580   565
  600   580
  620   600
```

BAN THE BOMBOUT

E. A. Anderson

Much has been written regarding the use of GET rather than INPUT to avoid dropping out of a program and still retain the editing features of INPUT. There is a very simple way to use INPUT and avoid dropping out of the program. Just open a file to the screen (OPEN#,3) at the beginning of your program; then add a CMD# before each input statement. Select the file number(#) to be unique from other files you may open later.

There are some other useful features of this approach. If a prompt string is printed on the same line as the entry the program will not advance when you hit RETURN unless an entry is made. If the prompt is made on the line before the entry the input will be zero for numeric entries and a null for string entries. The prompt question mark will not be present unless you include it in the prompt string. Multiple entries can be entered individually with RETURNs after each entry or with a comma separating each entry. After each RETURN the cursor will advance a space on the same line.

To exit the program do a LOAD/RUN (shifted RUN/STOP). It is best to use only string entries in any input statement to avoid dropping out of the program if the operator enters a string in response to numeric entry request. If LOAD/RUN is pressed when only string entries are used you will get a nice BREAK IN LINE #, almost as if it were designed to be used that way.

We have incorporated this feature in our latest versions of Personal Ledger, Data Logger and our new Omnifile data base software. We hope others will also use it to help Ban the Bombout.

Sample code and output

```
10 OPEN7,3
20 REM
30 REM YOUR PRO[illegible]
40 REM
50 CMD7,;:INPUT"NUMBER,NAME";N,N$:PRINT#7
60 PRINTN,N$
70 GOTO50

READY.
```

The ';' after 'CMD7,' and the 'PRINT#7' in line 50 are optional. They suppress the line feed after 'CMD7,' and add a line feed after the entry, respectively. Sample output is given below. An 'r' indicates a carriage return has been pressed. The boldface type indicates operator entries.

```
RUN

NUMBER,NAME? 1,JOHNr
  1        JOHN
NUMBER,NAME? 2,r
  2
NUMBER,NAME? JANE,r
  0        JANE
NUMBER,NAME? ,r
  0
NUMBER,NAME? 6rGEORGEr
  6        GEORGE
NUMBER,NAME? rrrrrrrrrrrrrr4,JILLr
  4        JILL
```

An alternative form of line 50 which gives identical results is:

```
50 CMD7,"NUMBER, NAME? ";:INPUTN,N$
   :PRINT#&
```

However, if either the prompt string or the ';' after the prompt string is removed, a zero will be entered for the number of a null for the string when RETURN is pressed after each entry as illustrated below.

```
50 CMD7,;:INPUTN,N$:PRINT#7
gives:

NUMBER,NAME?rr
  0

50 CMD7,"NUMBER,NAME? ":INPUTN,N$
   :PRINT#7
gives:

NUMBER,NAME?
rr
  0
```

ABBREVIATING BASIC WORDS

As explained in the instruction manual, any BASIC word takes up 1 byte of memory storage space. It has been stated that

the word "PRINT" can be abbreviated to "?" which saves time on entering programs. When listed, the word is expanded to its full form. Both forms take 1 byte per word.

We now have information on how to abbreviate the complete list of BASIC words. The algorithm is as follows :

1. For any BASIC word, type in the first letter of the word (e.g. V for VERIFY).

2. Hold down the 'Shift' key and type in the second letter. If you are in graphics mode, this will appear as a graphic character (e.g. for E). It is a good idea to go into lower case mode as the two letters are then easy to read.

In some cases, this two-letter method gives a possibility of more than one BASIC word (e.g. READ and RESTORE). For one of the words (usually the longer) it will be necessary to type the first two letters and the shifted third. All these abbreviations are converted to full words upon the command LIST.

Below is a complete list of the words and abbreviations:

BASIC	ABBREV
LET	lE
READ	rE
PRINT	?
PRINT#	pR
DATA	dA
THEN	tH
FOR	fO
NEXT	nE
DIM	dI
END	eN
GOTO	gO
RESTORE	reS
GET	gE
GOSUB	goS
OPEN	oP
CLOSE	clO
SAVE	sA
LOAD	lO
DEF	dE
RETURN	reT
STOP	sT
STEP	stE
INPUT#	iN
SGN	sG
ABS	aB
SQR	sQ
RND	rN
SIN	sI
ATN	aT
EXP	eX
AND	aN
NOT	nO
VAL	vA
ASC	aS
CMD	cM
VERIFY	vE
RUN	rU
CLR	cL
LIST	lI
CONT	cO
FRE	fR
TAB(	tA
SPC(	sP
PEEK	pE
POKE	pO
USR	uS
SYS	sY
WAIT	wA
LEFT$	leF
RIGHT$	rI
MID$	mI
CHR$	cH
STR$	stR

HOT TIPS

Paul Higginbottom

First I would like to answer some frequently raised questions about screen formating of data and then take a look at a few techniques to make a program smaller and more elegant.

Formatting numbers on the screen can cause problems when the TAB function is used. If a number is to be printed within a box, then it would be nice to ensure that the last digit of the number is always to the right hand side of the box.

For example, if the number is simply TABbed into the box and the number is '0', then it will appear to the left hand side of the box which does not look very smart. It is therefore necessary to use the length of the number (i.e the number of digits including the decimal point) to drive the TAB function. A number has a leading space and a trailing cursor right which need to be taken into consideration. The LEN function counts the number of characters that there are in the string. In order to use LEN it is first necessary to convert the number into a string variable using STR$. The number of characters in the number is given by:-

```
X=LEN(STR$(A))-1 .
```

- where A is the number. 1 is subtracted to take account of the leading space. So now, taking the above example, if we were to TAB(11-X) we would be in business!

Sometimes it is desirable to tack leading zero's onto integer numbers (i.e to display '0038' rather than '38'), but this is a little more tricky to program.

Let the number be S. Let S$ be the string version of S with leading zero's then:-

```
S$=RIGHT$("0000"+MID$(STR$(S),2),4)
```

MID$(STR$(S),2) converts S into a string without the leading spaces since it takes

the string version of S from the 2nd character onwards. Then the four rightmost characters of the string of zero's plus the shortend string version of S are concatenated to form S$.

* * *

Programs can be shortened a great deal with a little thought and an active imagination. For example it is often necessary to set a flag if a condition is met or to complement the flag. The usual code for complementing a flag is:-

```
1200 IF G=0 THEN G=1: GOTO1220
1210 G=0
1220 .......
```

On consideration, the statement:-

```
1200 G=1-G
```

- will be seen to have the same effect.

* * *

The program which follows is a further example of compact coding.

This is a screen dump routine which makes it possible to copy the screen onto the printer at any time. For example a graphics display on the screen can be transformed into hard copy by means of a GOSUB routine.

When a screen dump is performed it is necessary to read the screen and then turn it into a printable format. The easiest way to read the screen is to use the PEEK command, but the printer requires ASCII codes which are different from the PEEK/POKE codes and so a conversion is necessary. Now this can be achieved by a series of IF...THEN... statements but the result is rather long-winded:-

```
 5 REM CHARACTER SET
 6 PRINT"□"
 10 FORI=0TO255:POKE32768+I,I:NEXT
 15 PRINT"■■■■■■"
 50 REM * QUICK AND DIRTY SCREEN PRINT *

 100 OPEN4,4: PRINT#4,"": OPEN3,4,6
 102 PRINT"■■N■ORMAL LINE FEED OR ■C■OS
    "ED UP?":FOR I=1 TO 10: GETA$: NEXT
 103 GETA$: IF A$="" GOTO103
 104 N=18: IF A$="N" THENN=24
 105 PRINT#3,CHR$(N)
 110 FOR I=0 TO 999
 130 P=PEEK(32768+I)
 134 GOSUB500
 135 IF P<64 THEN P=P+64: GOTO200
 140 IF P<126 THEN P=P+128: GOTO200
 145 IF P<128 THEN P=P+64: GOTO200
 150 IF P<191 THEN P=P-64: GOTO200
 155 IF P<255 THEN P=191
 200 PRINT#4,CHR$(P);
 220 X=X+1: IF X=40 THEN PRINT#4,"": X=0
     : F=0
 240 NEXT
 250 PRINT#4,"":CLOSE4
 260 PRINT#3,"":CLOSE3
 300 END
 500 REM REVERSE FLAG
 515 IF P>127 THEN F=1: PRINT#4,"■";:
     GOTO600
 520 IF F=0 AND P<127 THEN600
 530 IF P<127 THEN F=0: PRINT#4,"■";
 600 RETURN
```

Line 10 prints out PET's upper case and graphics characters onto the screen to provide a test. It should be omitted when the program is used properly. Line 15 can have the number of 'cursor downs' adjusted so that the 'normal line feed...' prompt in line 102 does not overprint a crucial area of the screen. In a situation where the whole of the screen is required this part of line 102 should be removed altogether.

The main code conversion routine lies between lines 110 and 240; the subroutine 500-600 deals with the reverse field characters; lines 100-105 allow the line feeds to be closed up for printing graphics. Lines 200-255 ensure the printer buffer is emptied at the end of the program.

Now lets have a closer look at the difference between the PEEK/POKE codes and ASCII. If one considers the binary representation of a number of different characters is will become apparent that only the 3 most significant bits (bits 5,6,7) are changed. "Aha, a bit of BOOLEAN ALGEBRA will solve the problem!" Using the AND/OR functions is is possible to convert PEEK/POKE codes to ASCII in just one statement.

Thus if A=PEEK(32768), the top left hand corner of the screen, then:-

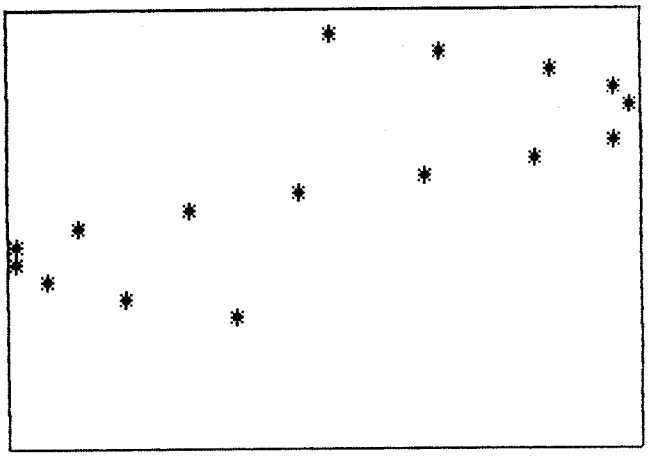
```
B=(A AND 128)OR((A AND 64)*2)OR((64-A
AND 32)*2)
```

- where B is the corresponding ASCII code.

The next matter to take care of is that the screen is 40 characters wide and the printer 80, so it is necessary to check when 40 characters have been printed and then send a carriage return. This can be done by considering the screen as 1000 address locations from 1 to 1000 and then looking at the particular screen address and seeing if it is divisible by 40.

Here is the new program:-

```
 5 OPEN6,4,6:PRINT#6,CHR$(18)
 10 OPEN4,4 :PRINT#4," ──────────────
    ──────────────────────": PRINT#4," |
    ";
 20 T=40: FORI=32768TO33767:A=PEEK(I):B
    =(AANDI27)OR((AANDI64)*2)OR((64-AANDI3
    2)*2)
 30 PRINT#4,CHR$(B);:IF(I-32807)/T=INT((
    I-32807)/T)THENPRINT#4,"| "CHR$(13)" |
    ";
 40 NEXT:PRINT#4,"──────────────────────
    ──────────────":PRINT#6,CHR$(24)
```



The lines are there to put a box around the display.

This routine will not output reverse field characters but this can be fixed quite easily because RVS characters have a PEEK/POKE code greater than 127 and so (A AND 128) will be 128 if the character is in RVS and Ø if not. Thus if before each character is output we use the following CHR$(146 - AANDl28) this will put out an RVS ON character if the screen is in reverse field mode and an RVS OFF character if the screen is in normal mode. RVS stays on until a carriage return or a RVS OFF is sent. The only bug I have noticed with this fix is between quotes (the RVS ON/OFF characters are printed literally), this can be avoided by remembering the position on the line and then sending a carriage return without line feed - CHR$(141) - then returning to the same point on the line and continuing.

I hope this whets your appetite for doing some exploration of your own. If you do then please send your discoveries to the Editor so that we can all share them via the medium of CPUCN.

COMPUTED GOTO

Have you ever wanted to program a GOTO followed by an expression such as:-

```
120 IF ST GOTO (ST * 10)
```

Normally the PET does not allow this but Brad Templeton has submitted a machine language routine that will handle a computed GOTO. The program fits in only 12 bytes and can be placed in any part of memory where it won't get clobbered by BASIC. It accesses code in ROM and therefore has two versions, one for original ROM and another for upgrade ROM.

```
Original ROM    JSR CE11    Checks for
                            comma else
                            SYNTAX ERROR
                JSR CCA4    Evaluates
                            expression
                JSR D6D0    Integer? =>0
                            <=63999
                JMP C7A0    Jump to GOTO
                            routine with
                            result

Upgrade ROM     JSR CDF8
                JSR CC8B    Same as above
                JSR D6D2
                JMP C7B0
```

Because the program has no reference to itself, it can be placed anywhere, but for now we'll put it in the second cassette buffer at 826 or hex Ø33A. Syntax for using the routine will be

```
                SYS826,expression
or...   G%=826:SYSG%,expression

e.g.  IF ST THEN SYS G%, ST* 10
```

BASIC Loader

With the following modification, both of the above routines can be loaded into the second cassette buffer and PET will decide which to use. This way programs using the computed GOTO can be run witheither ROMS.

```
                LDA $F202
                BMI $0D
 N.ROMS         JSR $CDF8
                JSR $CC8B
                JSR $D6D2
                JMP $C7B0
 O.ROMS         JSR $CE11
                JSR $CCA4
                JSR $D6D0
                JMP $C7A0
```

The following BASIC program will load the above code.

```
100 FOR J=826 TO 854
110 READ X
120 POKEJ,X
130 NEXT
200 DATA173,2,242,48,13,32,248,205,32
210 DATA139,204,32,210,214,76,176,199
220 DATA32,17,206,32,164,204,32,208
230 DATA214,76,160,199
```

Test the program with the following:

```
10 G%=826
20 PRINT"THIS IS A TEST"
30 SYSG%,2*10
```

PROGRAMME OVERLAYS ON A PET

Mike Stone

1. The 8K core of a PET is not usually a limitation in the home computer and hobbyist world, nor even in an educational environment where students are just creating small exercise programs. With the devices now available for attachment - the second cassette, the printer, and floppy discs shortly - the PET becomes a valid and genuine data processing machine, and complex string handling programs with files may well run out of space.

2. Programmers with experience on other computers know that one answer to this kind of problem is to break the program down into segments, so that only part of it is occupying memory at any time, and all or part is "overlaid" by other segments as required. When the program segments are on a disc, direct access features normally permit great flexibility, in that any required segment can be loaded; for tape only systems, the segments have to be arranged in order of need - e.g. job initialisation, main coding, and termination segments.

3. Since PET's BASIC includes a LOAD instruction to acquire dynamically a new program from tape, and (provided the new program is no longer than the one issuing the LOAD) all data areas remain available to the loaded program, the basis exists for an overlay system. However, for true overlaying, it is essential that some of the original program (e.g. control of the program flow, common subroutines, etc.) be retained throughout, whatever new segments are loaded. PET does not do this automatically; this paper tells you how to do it.

4. PET stores BASIC programs in location 1024 upwards, in the format for each instruction shown in Newsletter No. 0 page 4. Note that the pointers, and line numbers, are pairs of bytes giving low/high. The high must be multiplied by 256 and added to the low to give the actual quantity.

5. Whenever a line of code is entered from the keyboard, PET moves every statement around as necessary and re-adjusts all the chaining, so that statements are always stored in strict sequence of line number.

6. When your program contains a LOAD statement, this does NOT imply either CLR or NEW. The new program is simply loaded in at (and then executed from) location 1024, for as much space as it needs. The new program does NOT (as with some BASIC's) just replace those statements with identical line numbers; it is strictly a new program in its own right. However, any program statements at the end of the LOADING program whose space is not required by the LOADED program do remain unscathed by the LOAD. The problem is that the new program has no (forward-) chain into them, so PET knows nothing about them.

7. It follows from the above that if we code the instructions-to-be-preserved with high line numbers, and if the space needed by the newly-loaded segment does not over-write them; and if we can force the new segment to chain into the old instructions; then we have a real overlay system. So, if during the original program you can find in memory the last statement not to be preserved, you know it forward chains into the next highest line number, i.e. the first of the statements you do want to be preserved. Then when the overlay arrives, you need only find its very last statement and replace its forward-chain by the one you previously found, and both new and old code form a continuous program.

8. A very simple illustration follows

Enter this program (do NOT put any spaces, except after line numbers):

```
10 A=A+1
20 GOSUB50
30 LOAD"NEWPROG"
40 PRINT A*2
50 RETURN
```

This is stored as follows ("PEEK" values):

```
1024)   0
   5)  11   )   forward chain; 4x256
   6)   4   )   =1024+11=1035
   7)  10   )   line number 10
   8)   0   )
   9)  65       A
1030) 178       =
   1)  65       A
   2) 170       +
   3)  49       1
   4)   0
   5)  19   )   forward chain; 4x256
   6)   4   )   =1024+19=1043
   7)  20   )   line number 20
   8)   0   )
   9) 141       GOSUB
1040)  53       5
   1)  48       0
   2)   0
   3)  34   )   forward chain; 4x256
   4)   4   )   =1024+34=1058
   5)  30   )   line number 30
   6)   0   )
   7) 147       LOAD
   8)  34       "
   9)  78       N
1050)  69       E
   1)  87       W
   2)  80       P
   3)  82       R
   4)  79       O
   5)  71       G
   6)  34       "
   7)   0
   8)  43   )   forward chain; 4x256
   9)   4   )   =1024+43=1067
```

```
1060)  50   )   line number 50
   1)   0   )
   2) 153       PRINT
   3)  65       A
   4) 172       *
   5)  50       2
   6)   0
   7)  49   )   forward chain; 4x256
   8)   4   )   =1024+49=1073
   9)  55   )   line number 55
1070)   0   )
   1) 142       RETURN
   2)   0
   3)   0
```

If we want lines 50 and 55 to be available to an overlay, the important information is the forward chain in line 30, i.e. locations 1043 and 1044.

9. To see how it works, SAVE"PROG" and leave the cassette Record and Play keys down.

Enter NEW; then the following (again, no spaces):

```
5 A=A*2
10 GOSUB50
15 STOP
```

LIST if you like, to confirm that there are no lines 50 and 55.
SAVE"NEWPROG".

Now rewind your tape, and press RUN. PROG will be loaded, will print "2", and continue up the tape. When NEWPROG has been loaded, you will get

10. The last line, 15, does not chain into the old line 50. But line 50 is still there, in location 1058 et seq. So, do this :

```
POKE1043,34
POKE1044,4
```

LIST - and behold, NEWPROG now includes line 50 and 55!

What we have done is to use what we discovered about the first program (last sentence of paragraph 9) to modify the second program.

11. How do you program all this to happen automatically? It is not at all difficult. Let us assume that the statements-to-be-preserved are at lines 5000 and upwards. So just before that, code this (NO SPACES) :

```
10 REM---SUBROUTINE STARTS AT 4997---
20 REM
30 REM N1 LOW  ADDRESS OF LINE 4997
40 REM N2 HIGH ADDRESS OF LINE 4997
50 REM
60 REM BASIC2 LOCATIONS ARE 119 AND 120
70 REM
4997 N1=PEEK(201)
4998 N1=PEEK(202)
4999 RETURN
5000 REM**************************
5001 REM* PROGRAM TO BE PRESERVED *
5002 REM**************************
```

N1 is the low address of line 4998
N2 is the high address of line 4999

(Locations 201,202 (New ROM 119, 120) always contain, during instruction execution, the address of the next instruction - strictly, the "0" between instructions.)

12. Now, just before your program wants to load in the overlay program, code this (spaces if you like!):

```
850 GOSUB 4997
860 REM LOW ADDRESS OF 4999 (4998 IS 14
    BYTES LONG)
870 N1=N1+14
880 REM ACTUAL HIGH ADDRESS
890 N2=N2*256
900 IFN1<256THEN920
910 REM ADJUST PAGE BOUNDARY
920 N1=N1-256
930 REM BC IS ACTUAL ADDRESS
940 BC=N1+N2+1
950 REM FORWARD CHAIN
960 Z1=PEEK(BC)
970 Z2=PEEK(BC+1)
980 LOAD"NEWPROG"
```

N1 Low address of 4999 (the length of 4998 is 14 bytes).
N2 Actual high address of 4999.
880-890 Adjust for low page boundary.
BC is now actual machine address of line 4999.
Z1 and Z2 hold the forward chain locations out of 4999.

13. As the first instructions of NEWPROG, the chain-adjusting must be done; The necessary code is very similar:

```
5 REM PROGRAM NAME: NEWPROG
10 GOSUB3997
20 N1=N1+14
30 N2=N2*256
40 IFN1<256GOTO60
50 N1=N1-256
60 BC=N1+N2+1
70 POKEBC,Z1:POKEBC+1,Z2
1000 REM************************
1010 REM*                      *
1020 REM* MAIN BODY OF NEWPROG *
1030 REM*                      *
1040 REM************************
3997 N1=PEEK(201)
3998 N2=PEEK(202)
3999 RETURN
```

BC is now actual machine address of 3999

14. It is worth just reiterating that the total size of the incoming overlay (irrespective of line numbers; just its size in bytes occupied) must be less than the total size of the instructions being overlaid.

OVERLAYING LARGER ON TO SMALLER PROGRAMMES

Dave Williamson

Mike Stone's excellent article on overlaying programs emphasised the need to have an overlay program smaller (byte-size) than the program being overlaid. One important reason for this is that normally a larger overlay would trample on the variable table used by the first segment of the program.

Some long programs can be conveniently divided into an initialising segment and main segments with no common BASIC code, just the need to pass variables from one to the other. The following example illustrates how this can safely be accomplished even if a later segment is longer than the first.

Load the longest segment and determine the location of its final statement (if you RUN it you'll need to CLR variables first):

e.g. ? 8192 - FRE (0)

Let's say the answer is 7275. I prefer to give myself a little room to adjust the segment length in final debugging. So, add 10 bytes and convert the result to base 256:

e.g. 7285 = 28 x 256 + 117

The first statement of the first segment to be loaded should then be set thus:

e.g. 1 POKE 124,117:POKE 125,28:CLR

Locations 124 and 125 (New ROM 42 and 43) indicate the start of the variable table. The CLR statement adjusts all the other variable location markers in line with this as necessary.

And of course the last operation of the first segment should be to load the second. Since the segments do not need to share coding, no chaining is necessary.

THE 'ULTIMATE' SCREEN SAVE

Roger Davis

Many ways have been seen regarding how to effectively save a screen display as a file suitable for retrieval later.

Initially, I started by PEEKing every location on the screen, obtaining its value; then creating a sequential disk file of those values. On requiring that screen display all we have to do is open the file and POKE the data (starting at 32768) onto the screen. This is effective but takes quite a long time, and of course creates a large file (approx 14 blocks on the disk).

Next, I converted all the PEEK values into strings; this has the advantage of only requiring a small file (approx 5 blocks), and also will, when read print back much quicker because of the length of strings themselves. However, the conversion routine is complicated, takes time and I was sure there was a better way.

Whilst 'hacking around' I realised that if the screen is an area of memory, (which it is of course) there should be no reason why the screen display itself could not be saved as a program. To test this I simply jumped into the monitor with an 'SYS 1024' and then saved the screen display with:-

.S"SCREEN",08,8000,83E7

The current screen display was saved as a program file called 'SCREEN'. When I loaded 'SCREEN' it automatically displayed the screen display at the time it was saved in much the same way as Dos Support 4.0 will display the disk directory without replacing any program in memory. Great; there was only one problem, this system had saved everything I had typed onto the screen and of course scrolled off the top four or five lines of the screen as well!! Nevertheless, the idea was worth persuing, so I created the following machine code 'SAVE' routine which at any time during a program will, with a SYS call, save exactly the screen display. This program can be used in fact to save any area of memory, with a given file name to any device.

This worked OK when tested, however, if a file created in this way is loaded from a

BASIC program, the 'calling' program will RUN immediately after the new screen is displayed. Depending on the application required, this can be avoided as shown in the following (simple!) example:-

```
10 REM TEST 'SAVE' PROGRAM
20 STOP
30 LOAD"0:SCREEN",8
```

Type RUN30 to try this.

Another way of doing this is to change line 20 to "20 GOTO 40". There is no point in putting an END after the LOAD instruction as the program will not reach any line number after the LOAD.

The same system of operation can be used for a 'load' as shown in the enclosed program.

On a slightly different topic, with so many programs being written for Commodore in machine code or a combination of machine code and ASCII, it is sometimes difficult to find the start and end address of the program. This can be easily done in the following way:-

For disk:
1) enter'open8,8,8,"program name",p,r'
For tape:
1)enter'openl'
2)enter'sys 1024'
3)enter'm 027a,0290'

The resulting memory display is as follows:-

027a = the type of file
027b-027c = starting address
027d-027e = ending address
027f- = file name.

```
SCRSAVE.SRC......PAGE 0001

LINE# LOC   CODE        LINE

0001  0000                          ;***************
0002  0000                          ;*             *
0003  0000                          ;* SAVE ROUTINE *
0004  0000                          ;*      BY     *
0005  0000                          ;*  ROGER DAVIS *
0006  0000                          ;***************
0007  0000
0008  0000             BEGIN  =$033A
0009  0000             SAVE   =$F6AD
0010  0000             READY  =$C38B
0011  0000             DEVICE =$08
0012  0000             LOSTAR =$00
0013  0000             HISTAR =$80
0014  0000             LOEND  =$E8
0015  0000             HIEND  =$83
0016  0000             SECON  =$D4
0017  0000             LONAME =$67
0018  0000             HINAME =$03
0019  0000             NAMLEN =$09
0020  0000                    ;
0021  0000                    *=BEGIN
0022  033A
0023  033A  A9 4C             LDA #$4C
0024  033C  8D FD 03          STA $03FD      ; PROTECT?
0025  033F  A9 08             LDA #DEVICE    ;GIVE DEVICE NO.
0026  0341  85 D4             STA SECON
0027  0343  A9 67             LDA #LONAME    ;LO ADDR OF FILENAME
0028  0345  85 DA             STA $DA
0029  0347  A9 03             LDA #HINAME
0030  0349  85 DB             STA $DB
0031  034B  A9 09             LDA #NAMLEN    ;LEN OF FILENAME
0032  034D  85 D1             STA $D1
0033  034F  A9 00             LDA #LOSTAR    ;LO START ADDRESS
0034  0351  85 FB             STA $FB
0035  0353  A9 80             LDA #HISTAR
0036  0355  85 FC             STA $FC
0037  0357  A9 E8             LDA #LOEND     ;LO END ADDRESS
0038  0359  85 C9             STA $C9
0039  035B  A9 83             LDA #HIEND
0040  035D  85 CA             STA $CA
0041  035F  A2 00             LDX #00
0042  0361  20 AD F6          JSR SAVE
0043  0364  4C 8B C3          JMP READY
0044  0367                    .END
```

PROGRAM NAME: SCREEN

```
100 REM
110 REM*** SCREEN SAVE ROUTINE ***
120 REM    BASIC SEGMENT.
130 REM    BY ROGER DAVIS.
140 REM
150 REM
160 REM*** GET DRIVE NUMBER
170 REM
1000 INPUT"DRIVE NUMBER";D%: IF D%<>0
     AND D%<>1 GOTO1000
1010 OPEN15,8,15,"I"+STR$(D%): POKE871,
     D%
1100 REM
1110 REM*** GET FILE NAME
1120 REM
1130 INPUT"FILENAME";F$: L=LEN(F$): IF
     L=0 GOTO1130
1140 POKE844,L
1150 FORI=1TOL: A=ASC(MID$(F$,I,1)):
     POKE872+I,A: NEXT
1200 REM
1210 REM*** PROGRAM START AND END
1220 REM
1230 INPUT"START ADDRESS (32768)";S
1240 INPUT"END ADDRESS   (33768)";E
1250 D=S:GOSUB1300:POKE848,R%:POKE852,H
     %
1260 D=E:GOSUB1300:POKE856,R%:POKE860,H
     %
1270 REM
1280 REM*** SAVE MEMORY
1290 REM
1295 END: SYS826: END
1300 REM
1302 REM*** GET HI AND LO
1305 REM
1310 H%=(D/256):R%=D-H%*256:RETURN
```

Dave Middleton

Rogers solution to saving the screen as an area of memory using machine code prompted me to perform the same operation with BASIC. Lines 1010-1030 are the actual save routine. The rest of the program consists of examples of how it works and how to use it. The only fault with my version of the program is that the name of the file to be saved can not be changed. There are two solutions to this:-

1. POKE the filename of the screen to be saved into the actual program.
2. SAVE the program under the name 'SCREEN' and then rename the file on disk.

Like DIMP screen save is a solution looking for a cause but I had great fun playing with it.

```
1000 INPUT"DRIVE NUMBER";D%: IF D%<>0 OR
     D%<>1 GOTO1000
1010 OPEN15,8,15,"I"+STR$(D%): POKE871,D%
1100 REM
1110 REM GET FILE NAME
1120 REM
1130 INPUT"FILENAME";F$: L=LEN(F$): IF L
     =0 GOTO1130
1140 POKE844,L
1150 FORI=1TOL: A=ASC(MID$(D$,I,1)):
     POKE872+I,A: NEXT
1200 REM
1210 REM *** PROGRAM START AND END
1220 REM
1230 INPUT"START ADDRESS (32768)";S
1240 INPUT"END ADDRESS   (33768)";E
1250 D=S:GOSUB1300:POKE848,R%:POKE852,H%
1260 D=E:GOSUB1300:POKE856,R%:POKE860,H%
1270 REM
1280 REM SAVE MEMORY
1290 REM
1295 SYS826: END
1300 REM
1302 REM GET HI AND LO
1305 REM
```

Machine Code Programming

MACHINE CODE

When the PET was first introduced the fact that it could be programmed in machine code was virtually unknown. Now of course most dedicated programmers learn machine code after exploring the intricasies of BASIC. A lot of programs are sent to us written in machine code because BASIC is just not fast enough to perform the task required or the function is not present in the interpreter. Many programs mix BASIC and machine code, allowing BASIC to handle the number crunching and letting machine code print the results onto the screen for example.

Learning to program in machine code can be a very traumatic experience because it lacks the nice friendly error messages which BASIC gives out. The only time you really know that your program has not worked when programming in machine code is when the PET crashes!

Learning to program in machine code also teaches a programming skill which many BASIC programmers forget, this being planning. It is not possible to write machine code by sitting down and just typing it in. Working with such small commands (loading the accumulator with a byte is about as small as you can get!) means that it is necessay to plan out every algorithm before entering it into the PET.

Using an Assembler system simplifies the task considerably. The editor allows the user to type in code in a manner which is user friendly, ie. you type in english rather than Hexadecimal! Code can be modified with ease and it is also possible to understand the program when it is read at a later date.

A powerful assembler has a drawback when being used by a beginner because it takes quite a long time to save source code with the editor, load/run the assembler wait for it to produce the errors it detects in the source code (the english), re-run the editor and correct the errors. Finally the source code will be passed by the assembler as being error free. The machine code can now be run and the most likely result will be that the PET crashes due to an error in the program logic. To debug these errors is probably the most difficult part as it now means delving into the Hex code stored in RAM to try and find out where the logical error occured.

Debugging aids are available, Supermon is an excellent aid for debugging machine code, especially the single step facility which allows you to run a machine code program a single instruction at a time. The program will still crash but at least you will know where it crashed!

Most of the articles in the next section introduce machine code. Only those by Paul Higginbottom attempt to cover the topic to any depth. It is very difficult to know where to begin but in the next few pages you will get a good selection of starting points!

If machine code programming is going to be undertaken it is probably worth invesing in a reset key this will allow you to uncrash most programs. It will however only work with New ROM PETs as the NMI used to cause the reset is tied high on the 8k PET and thus can not be accessed. For a really cheap reset key see 'Hardware Reset'.

I learnt machine code by using the comprehensive MOS Technology Programming Manual, available from us at Commodore, and also by performing a great deal of experimentation. If you prefer detailed instructions then read 'Programming the 6502' written by Rodney Zaks and published by Sybex; the book is reviewed elsewhere.

MACHINE CODE ENVIRONMENT

If you wish to write machine code programs in your PET and do not wish to have BASIC trampling all over them here is a suggestion:

When the PET is first powered up a test pattern is written into and read back from the RAM in ascending address order. When this routine discovers a location which does not read back properly it presumes that it has run out of RAM and displays XXXX bytes free. At this point it makes a note of where it thinks the 'top of memory' is. A quick glance at the memory map will show that BASIC program text is stored from location 1025 upwards and strings are stored from the top of the memory downwards which means that in any normal circumstances there is nowhere in the PET main memory where you can hide your machine code routines.

If however, the first thing you do after

powering up the PET is to alter the top of memory pointer to say 6000 everything from 6001 upwards, as far as PET is concerned does not exist. e.g. strings will be stored from 6000 downwards etc. and machine code programs can be safely put in location 6001 upwards. This pointer is held in locations 134 and 135 constituting a 16 bit pointer with 134 being its lower 8 bits. This is a binary pointer which means that we must convert your 6000 or whatever to binary before POKING locations 134 and 135 with th information. In the standard 8K PET 134 will be 0 and 135 will be 32 (32 x 256 =8192). Remember that 1025 bytes are used for house keeping by the PET (8192 - 1025 = 7167). However, to give the PET a ceiling of 6000 we convert 6000 into binary which gives us POKE 134,112 and POKE 135,23.

This issue we look at fundamental concepts; different kinds of memory; addressing and jumping to PET's built in machine code routines using the SYS command. Addresses are given for both BASIC1 and BASIC2 PET's, the latter in brackets.

ADDRESSING

Every memory location in your PET contains one byte of information. In order for PET to get at these bytes it must have a means of accessing them. Therefore each and every memory location has its own individual address; all 65536 of them. The microprocessor places these addresses on the address buss which immediately enables one memory location to the data buss. Bearing in mind, one of two operations can happen now. PET can either place a byte into that location (i.e. POKE) or "look" at what's already there (i.e. PEEK). When performing the first operation the microprocessor places a byte on the data buss and transfers it along the buss and into the enabled memory location.

In the second operation, the information or byte in the enabled location is transferred onto the data buss and along the data buss back to the microprocessor. This location is not "emptied" but rather only a duplicate or copy of the information is transferred. Once either of these operations is complete the microprocessor then places a new address on the address buss and another location is enabled. This process repeats thousands of times every second, however these operations aren't possible on all memory locations, but I will explain this later.

The microprocessor has control of 99.9% of the addresses being placed on the address buss. That extra 0.1% control was left for the user and can be obtained through use of the PEEK, POKE and SYS commands. When executing these commands the user must choose an address. This address will be one of the 65,536 memory locations (i.e. 0 to 65,535). This is where the memory map enters the picture. The memory map may well be your most powerful tool for choosing addresses. If you look at the map you will see that all of the addresses are listed in ascending order down the left hand side; first in hexadecimal and then in decimal. (See section on hexadecimal and binary for explanation of this conversion) the decimal address is the one you use when executing the above 3 BASIC commands. To the right are the descriptions of what you can expect to find at the corresponding addresses. If we then PEEK these addresses we are returned the actual bytes that are in those particular memory locations. For example, let us say during a program we hit the STOP key and got:

```
BREAK IN 600
READY.
```

PET gets '600' from a storage register at addresses 138 and 139 (54 and 55). We could also PEEK these locations and find that 600 is indeed stored in 138, 139 (54, 55). However it is not stored as a six, a zero and a zero. Instead it is stored as the decimal conversion of the line numbers representation in hexadecimal. All information of this type is returned in this manner. Now that we know what the memory map will help us do let us cover some of the rules.

RAM AND ROM

We all go through life with basically 3 types of memory:

1. MEMORY PRESENT: This memory we use to remember things like what street we're driving on or our present location.

2. MEMORY PERMANENT: Things like our names and fire is hot we never forget.

3. MEMORY PAST: Recent occurences and not so recent such as things we did 10 or 12 years ago.

In the PET there are only two:

1. RAM Random Access Memory: this type of storage is used for our programs and things that change such as the clock updating routines and loading routines to name a few. These functions would have to be programmed into PET on each power up if they weren't permanently 'burnt in'.

The third type, memory past, is instantly 'forgotten' on power down. The only way to recall it is to first save it on tape, discs, etc.

Recall earlier I mentioned that POKE and PEEK are not possible on all memory locations for several reasons:

A. Not all PET memory locations actually exist. On the memory map, locations 1024 to 32767 is the 'available RAM including expansion'. If you have a PET with 8K, simple arithmetic shows that 3/4 of the available RAM space is non-existent. If you decide to expand your system, PET will 'fit' the added RAM into this area. However, POKing or PEEKing this space (i.e. 8192 to 32767) will return invalid results on 8K PETs.

B. The same concept applies to locations 36864 to 49151. This is the available ROM expansion area.

C. Next on the memory map is the Microsoft BASIC area; locations 49152 to 57463. This is the memory that recognizes and performs your commands. Changing the contents of these locations in impossible because it is Read Only Memory and is actually 'burnt in' at the factory. Therefore, POKing these locations will simply do nothing. Also, Microsoft requested that these locations return zeros if PEEKed (for copyright reasons). This last point does not apply to BASIC2 PETs.

With these three rules and your memory map you are now equipped to explore capabilities of your PET that you probably never thought possible. Before we try some examples let's go into one more important occurrence that may have had you scratching your head since that first power up.

MISSING MEMORY

When you turn on your 8K (where K = 1024) PET, the first thing it tells you is 7167 BYTES FREE; a reduction of almost 12%, similarly a 32K PET displays only 31743 BYTES FREE.

Q. Where did the missing 1024 bytes go?

A. It's still there...right below the available RAM space (notice it starts at location 1024). PET uses this memory to do some very useful operations for you which you can find and access by looking them up on the memory map.

Q. But why not do this in ROM space?

A. PET needs RAM type memory to store this data because it is always changing. The information in this "low" end of memory is actually produced by routines found in ROM.

Take for example the built-in clock. The clock or time is stored in locations 512 and 514 (141, 142 and 143) in RAM. However, the data comes from a routine found in ROM. The time is of course always changing, therefore it must be stored in RAM. But because it is in RAM you may also change it; either by setting TI or TI$ or you can POKE the above three locations. Try it.

Now let's try some examples;

1. Location 226 (00E2 in HEX) holds the position of the cursor on the line. Try these:

```
POKE 226,20:?"PRINTS AT NEXT
     SPACE

?"123456789";:?PEEK(226)

(BASIC2)
POKE 198,20:?"PRINTS AT NEXT
     SPACE

?"123456789";:?PEEK(198)
```

2. Location 245 (216) stores the line the cursor is presently on (0 to 24). POKing this location will move the cursor to the specified line after a display execution. For example try;

```
?"A":POKE 245,10:?"B":?"C"

POKE 245,21-1:?"cu":POKE 226,20
     :?"PRINTS HERE"

(BASIC2)
?"A":POKE 216,10:?"B":?"C"

POKE 216,21-1:?"  ":POKE 198,20
     :?"PRINTS HERE"
```

The above will move the cursor to line 20 (21-1), print a 'cursor up' on line 21 and display your message starting at column 21, line 20..

While experimenting with out-of-range values I obtained some rather interesting results. Try POKing location 245 (216) with a number greater than 24, say 40 or 60, and hit the cursor up/down key a number of times. Also experiment with unusual numbers in location 226 (198) such as;

```
POKE 226,100:?"123456789"

(BASIC2)
POKE 198,100: ?"123456789"
```

3. Location 526 (159) is the reverse field flag. POKing this address with a non-zero value will execute the following same line print statements in RVS field. Once finished, PET resets 526 (159) to zero.

Try this;

```
POKE 526,1:?"123":?"456"

(BASIC2)
POKE 159,1:?"123":?"456"
```

now INST a semi-colon between 3" and the colon (i.e. ...23";:?"4...) and re-execute.

BEGINNING MACHINE CODE

Paul Higginbottom

This series of articles form a guide to machine code programming on the PET, starting from first principles, and going on to explain how routines from the PET's own operating system can be used.

First, let's look at the heart of the microcomputer - the microprocessor.

A microprocessor relies on the fact that a voltage can be used to make a line into the processor 'high' or 'low'. In the case of the PET that means a line is either at 5 volts or it is grounded. These two states can be used to represent a '1' or a 'Ø'. Or, if you like, 'ON' and 'OFF'. Since binary, or base 2 uses only the digits Ø and 1, the microprocessor has great significance in arithmetic applications.

Early programming was done by telling the microprocessor which lines to hold high and which to hold low. This is called micro-programming, and was very tedious. Soon, as expertise grew, a new level of programming became possible, where actual instructions could be used that were based on hardware architecture. For example, an instruction to enable a central store or register to be loaded with the present contents of a memory. Each memory location being defined as a specified number of ON's and OFF's like a binary number of a specified number of digits. Each 'ON' or 'OFF' digit is known as a 'BIT' and each actual memory location is called a 'BYTE' (a series of 8 'bit's).

Present day microprocessors have the basic capabilities of understanding an instruction set that can be used to manipulate data and move data around to different memory locations. This level of programming is called machine level programming (thus - machine code).

The operating system of the PET, (which includes the BASIC language, the screen editor, and the peripheral input/output routines etc.) is in fact, one huge machine code program that does all of the memory manipulation for you. It also evaluates all of the BASIC commands. The memory that it uses can be divided up into two catagories:

1) ROM (Read Only Memory)
2) RAM (Random Access Memory)

The ROM chips inside the Pet contain the routines that never need to change, and so are 'burnt' into the chips. If these routines weren't 'burnt' in, then as soon as the machine was switched off, all of the contents of the ROMs would be lost.

The RAM chips retain what ever bits have been set by the microprocessor and can thus be changed by it, only for as long as the computer system remains switched on. The microprocessor actually manages all the memory manipulation, so that, for example, when a line of BASIC is typed in it knows what area of memory to change.

Let's take a closer look at BASIC. When a BASIC program is run, each instruction is examined by the PET, and the corresponding piece of machine code in the operating system is executed. The operating system INTERPRETS them for the microprocessor. Thus, if Commodore chose to produce PETs with a different language all that would need changing is that part of the operating system that can interpret user programs.

Although microcomputers generally interpret and execute BASIC programs step by step, remembering things like where to jump back to after a subroutine has been finished, and where the start of the FOR NEXT loop is, this is not always the case. Often a COMPILER is used. This is a machine code program that actually turns the whole program into machine code! Anyway, let's just remember that the PET interprets BASIC, and executes the necessary routine(s) in the operating system.

If you examine a line of program written in BASIC; you will see that it consists of a line number, a statement with the necessary number of operands that instruction requires, and maybe another instruction, separated from the last by a colon. This is analogous to machine code, where a machine code instruction is stored as a number in a byte of memory, followed by the necessary number of operands (bytes of data) for that instruction. This represents either a number, or an address of a memory location inside the Pet that the program needs to access. The next machine code instruction follows on directly.

The trouble is that to look at it, machine code is just a stream of numbers stored in memory. Each of these numbers can be checked in turn using the PEEK instruction available from BASIC, or they can be inspected using the Machine Code Monitor (this utility is part of the ROM routines in the BASIC2 PETs, and is available as a program which can be loaded into RAM on the BASIC1 PETs.

Some way of representing machine code is needed to make it easier to use. This is done by using a three letter mnemonic associated with each instruction and a standard notation for the operand(s) associated with each instruction. So we need a program (BASIC or machine code) to enable us to create and edit a program in machine code, not as a series of numbers, but like BASIC for example, with reserved words for each instruction. Such a program is called an 'Assembler'. This allows the programmer to write and edit machine code programs in mnemonic format.

When the Pet is turned on, it

automatically jumps into a machine code routine which checks free memory space it has, and then prints on the screen '### COMMODORE BASIC ###' etc. From then on, it expects you to type in a BASIC instruction, and if it doesn't understand it, a 'SYNTAX ERROR?' is generated. So how do we get into machine code ? Well there are two BASIC instructions that provide a "bridge" from BASIC to machine code: One of them is 'SYS(X)', and the other is 'USR(X)'

The 'SYS' command requires a decimal location ('X'), and it will then start executing machine code from there. The 'USR' function has already had the address of where the machine code program starts, stored in two specific locations, and its paramater (X) is a number you can pass to the machine code program. This means that if, for example, a machine code program drew a square box in the left hand corner of the screen, then it is possible to pass the size of the box to the machine code program by making that the paramater put inside the USR command. For a fuller explanation of this command refer to the Pet User Manual supplied with your machine, or the "Pet Revealed" by Nick Hampshire which can be ordered through your local dealer.

So, to recap, machine code is a series of instructions followed by their respective operands, stored in a specific part of PET memory. To facilitate machine code programming some way of representing these instructions is needed, and so a complete set of mnemonics for all the instructions, and a standard notation has been devised. A special program is necessary to allow us to type and edit our programs using these mnemonics, which can be translated into the corresponding codes, and stored in memory. Such a program is called an Assembler.

A Disassembler reverses the process and allows an existing machine code program to be "translated back" into mnemonic notation. This is extremely useful when trying to decipher other people's machine code routines.

There are numerous assemblers and disassemblers on the market including a disk based assembler from Commodore. I use the Commodore assembler and find it to be about the best.

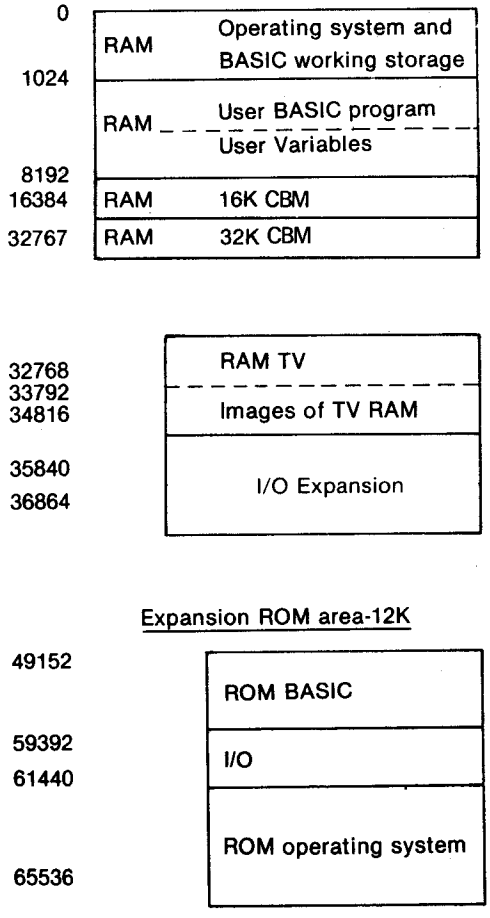
Having in this article examined the general features of machine code, I will in future articles look at the features of 6502 machine code. The 6502 is the micro-
processor used inside the PET. In the next article I will look at the way PET's memory is organised and how a memory location is addressed.

ADDRESSING

Paul Higginbottom

The 6502 uses 16 bits of memory to define the address of any byte of memory inside the PET. Consequently the highest address that PET can access, is the ammount represented by the largest possible sixteen digit binary number: 1111111111111111, which is 65535 in decimal. As a byte consists of eight bits and an address is 16 bits long then each address can be specified by two bytes, (apart from those special addresses 0 - 255, called zero page, which can be recognized by the 6502 with an address of only 8 bits). The rightmost eight bits or the lower half of the number is called the 'low byte' of the address and the upper eight bits or the second byte of the address is called the 'high byte'.

A single byte can range in value between 00000000 in binary (0 in decimal) to 11111111 in binary (255 in decimal). Memory is frequently referenced in 256 byte chunks (0-255). Each chunk is called a page of memory, and so the bytes with addresses 0-255 are called page zero (the 6502 has special zero page instructions), and the bytes 256-511 are called page one, etc. The PET uses pages 0,1,2,3 for it's own workspace; if we alter some of the values stored here(*), it is possible to make the PET operate in a different way. The table below outlines the way PET operating system uses its memory -



CBM memory map.

* A detailed memory map for this area (0-1024 decimal) appears at the end of this section. Also given are the ROM entry points for BASIC1 and BASIC2.

HEXADECIMAL NOTATION

This is the notation which is most frequently used by machine code programmers when referencing a number or address in an Assembly Language program.

Some assemblers make it possible to refer to addresses and numbers in decimal (base 10), binary (base 2), or even octal (base 8) as well as hexadecimal (or just 'hex'). Hexadecimal seems a bit difficult to grasp at first, but with practice it doesn't take long to master it.

If you look at a set of decimal (base 10) numbers, you will see that each digit in that number ranges between zero and a number equal to the base less one, i.e - 9. THIS IS TRUE OF ALL NUMBER BASES. Binary (base 2) numbers have digits ranging from zero to one (which is one less than the base). Similarly Hexadecimal numbers should have digits ranging from zero to fifteen, but as we do not have any single digit figures for the numbers ten to fifteen, so the first six letters of the alphabet are borrowed instead:-

Dec Hex

```
 0 - 0
 1 - 1
 2 - 2
 3 - 3
 4 - 4
 5 - 5
 6 - 6
 7 - 7
 8 - 8
 9 - 9
10 - A
11 - B
12 - C
13 - D
14 - E
15 - F
16 - 10
```

If that's confusing, let me try to put it another way:

```
    3   2  1  0
  10  10 10 10
--------------
1000 100 10 1
--------------
   4   5  6 9   = 4*1000+5*100+6*10+9
```

Example of how a base 10 (decimal number) is constructed.

```
    3   2  1  0
  16  16 16 16
--------------
4096 256 16 1
--------------
   1   1  D 9   = 1*4096+1*256+13*16+9
```

Example of how a base 16 (hexadecimal number) is constructed.

Therefore 4569(Base10) = 11D9(Base16)

The range for memory locations which the 6502 can address - 0 - 65535 (decimal) is 0 - FFFF in hexadecimal notation.

Usually hexadecimal numbers are prefixed with a dollar sign. This is to distinguish them from decimal numbers. Each address is only a four digit hexadecimal number ($0000 - $FFFF), and splitting the address into two byte format is easy as the leftmost two digits of a four digit hexadecimal address are the 'high byte', and the rightmost two digits are the 'low byte'.

This can be understood better if you think of binary again. Remember I said that the lower eight bits of an address constituted the 'low byte' and the upper eight bits were the 'high byte'. Well any 8 bit binary number can be uniquely represented by a two digit hex number. As the range of eight bits is 0 - 255, this is $00 to $FF in hex.

MACHINE CODE MONITOR

Let's now enter the machine code monitor. BASIC2 PET owners simply type:- SYS 1024 <RETURN>. Those with BASIC1 follow the instructions on the Commodore Machine Code Monitor tape available from your local dealer. Your PET's screen will clear then come up with something like the display below though the actual numbers may be different.

```
B*
     PC  IRQ  SR AC XR YR SP
.;  0401 E62E 32 04 5E 00 F6
.
```

Don't worry about these numbers for the moment, but type in:-

.M 033A 0360 <RETURN>

You will see rows of 9 Hex numbers. The first 4 digit one is the address of the first byte of memory being shown in that row, and the other eight numbers are the actual contents of the memory locations beginning at that start address.

I hope that now we have established the way that the PET accesses its memory, i.e as a sixteen bit number which is split in half as a lower and upper eight bits, we will soon be able to write our first machine code program!

In order to allow PET owners without assemblers to run the example programs associated with this "Beginning Machine Code" series of articles I will give the Hex listings as well as Assembly code

versions. But since the I will be concentrating on what the machine code programs doing then it will be useful for everybody following these articles to study the programs in their most 'readable form', i.e. in assembly source code.

SAVING A MACHINE CODE PROGRAM

Let us assume that you know your program resides in the second cassette buffer which starts at decimal location 826 ($033A in hexadecimal). Having entered the monitor -

for resident monitor type:-

```
.S "FILENAME",01,033A,0400 <RETURN>
```

for tape based monitor type:-

```
.S 01,"FILENAME",033A,0400 <RETURN>
```

Where 'FILENAME' is the name you wish to call the program, '01' is the device number you wish to save the program on, '033A' is the start address, and '0400' is the address up to which you wish to save the contents of memory. To save on the disk (if you have BASIC2) you would need to specify the drive number in the filename as usual, and make the device number as '08'.

LET'S WRITE A MACHINE CODE PROGRAM

Now that I have described addressing, and how to create and save machine code programs, I think that it is about time we wrote one.

The 6502 has a central register known as an accumulator, which many of its machine code instructions act upon. It also has two other registers, one is called the X register, and the other (surprise, surprise) is called the Y register. These two registers can be used in the same way as the accumulator in most circumstances, nut their main function is that they can be used as indexes which means that they can be used as offsets to a particular address.

For example, if you wanted to draw a line of A's across the top of the screen in BASIC by using the POKE command, you could do this by typing in the following line:

```
PRINT"<CLR>" ; : FOR I = 0 TO 39 : POKE 32768 + I , 1 : NEXT <RETURN>
```

"1" is the POKE code for an 'A'

In this instance, 'I', which is a BASIC variable is being used as the offset from the main address which is 32768 ($8000 in hex). In machine code we could use the X register. For example:-

```
ADDR HEX CDES MNE OPER.  COMMENTS
033A A9 01    LDA #$01   ;LOAD ACCUMULATOR WITH
                         ;THE VALUE '1'. THE '#'
                         ;SYMBOL SIGNIFIES THAT YOU WISH
                         ;TO LOAD THE ACCUNMULATOR WITH
                         ;THE 'VALUE' 1, RATHER THAN THE
                         ;CONTENTS OF MEMORY $01

033C A2 00    LDX #$00   ;LOAD THE X REGISTER
                         ;WITH '0'.

033E 9D 00 80 STA $8000,X;STORE THE CONTENTS
                         ;OF THE ACCUMULATOR AT
                         ;HEX ADDRESS $8000
                         ;(WHICH IS 32768 IN
                         ;DECIMAL) OFFSET BY
                         ;OFFSET BY THE VALUE OF
                         ;X WHICH IS ZERO IN THE
                         ;FIRST CASE.

0340 E8       INX        ;INCREMENT THE X REG.

0341 E0 28    CPX #$28   ;COMPARE THE X REGISTER
                         ;WITH HEX $28 (DEC 40)

0343 D0 F8    BNE $033E  ;BRANCH IF IT'S NOT
                         ;EQUAL TO $28 BACK TO
                         ;HEX LOCATION $033E
0345 60       RTS        ;OTHERWISE RETURN FROM
                         ;SUBROUTINE I.E EXIT.
```

The Hex listing is as follows:

```
.M 033A 0342
.:  033A A9 01 A2 00 9D 00 80 E8 <RETURN>
.:  0342 E0 28 D0 F8 60 00 00 00 <RETURN>
```

Having either typed the program in the monitor or created it by using an assembler, you can run it by doing a SYS 826 <RETURN> command. I think I will leave this machine code program with you to ponder over and we will go over it next time.

0	- 255	Pet Workspace.
256	- 512	Processor Workspace.
513	- 633	Storage area.
634	- 825	1st Cassette Buffer.
826	- 1017	2nd Cassette Buffer.
1024	- 32767	Basic program and variable area.
32768	- 36863	Screen storage space.
36864	- 49151	Nothing. Room for expansion.
49152	- 57592	Machine code Basic Interpreter.
57593	- 59391	Machine code to control the keyboard, screen.
59408	- 59411	PIA1 - Keyboard I/O.
59424	- 59427	PIA2 - IEEE Bus I/O (Peripherals).
59456	- 59471	VIA I/O and timers.
61440	- 65535	Power-up routines, diagnostic monitor.

Anyway that's enough for this newsletter. Next time we will deal with ?

INSTRUCTION MODES AND 6502 OPERATIONS

Paul Higginbottom

You remember last time I left you a program that I said I would let you ponder over, and that I would go over it this time ? Well here goes:

The program demonstrated many different principles of 6502 machine code programming, and instruction modes. These will be explained statement by statement.

line 1: LDA #$01 - This is an immediate mode instruction and 'says' :load the accumulator with the value $01 (the '#' symbol tells the assembler that is a value rather than the contents of memory location $01 that is to be loaded into the accumulator). Immediate mode means that the value to be used is immediately after the instruction, rather than the contents of a memory location defined by the operand.

line 2: LDX #$00 - This is also an immediate mode instruction and 'says' :load the X register with the value $00.

line 3: STA $8000,X - This is an absolute indexed mode instruction and says: store the contents of the accumulator at the address [$8000 + the contents of the X register]

line 4: INX - This is an implied mode instruction since the instruction itself defines what register the operation is being done to.

line 5: CPX #$28 - This is an immediate mode instruction and says: compare the contents of the X register with $28

line 6: BNE $033E - This is an implied mode instruction and says: branch if not equal to the address $033E. Thus, if the comparison of X and $28 (or decimal 40) was not equal, i.e X<>40, then branch back to the address $033E, and continue executing. The BRANCH instructions will be explained fully later in this article.

line 7: RTS - This is an implied mode instruction and says: return from subroutine. This instruction will cause execution of thois routine to cease, and return to where it was executed from. As this was a SYS call, then it will return to a usual BASIC 'READY' mode.

In the processor, there are many registers, some I have already mentioned. I think that now would be a good time to describe those registers.

ACCUMULATOR - This is the main eight bit register, around which most of the instructions are based.

X register - This is an eight bit index register. 'Index' means that it can be used to offset a base address by its contents. This was shown in the first program.

Y register - This is also an eight bit index register and has similar facilities as the X register. However some 6502 instructions are designed only for use with the X or the Y registers but not both.

Status register - This is possibly the most valuable and vital register in the processor. It uses each of its eight bits to describe a state that has arisen by one instruction, or by a sequence of instructions.

Each bit will be described below:-

N V B D I Z C

N - Negative Flag.

This is set if a negative condition is

made. This could be done with a subtraction, decrement, comparison or some other instruction. There are no instructions to set it directly. (The comparison instruction, does an internal subtraction and sets the flags according to the result of the subtraction). BMI (branch on minus) will take the branch if the negative flag is set.

V - Overflow Flag.

This is set if a result actually overflowed the register being operated on, e.g if a value is added to a register that makes it go over $FF (or 255 decimal), then the value in the register would be the result-256, and this might make mathematical calculations go wrong, and so the overflow flag is set. It may be cleared directly with the instruction CLV (clear overflow flag). It may be detected by the instructions BVC (branch on overflow clear) and BVS (branch if overflow set).

B - BRK command executed.

This is set if the BRK (break) instruction is executed. It would not be worthwile covering this at the moment.

D - Decimal mode flag.

This is set and cleared by SED and CLD (set, and clear decimal mode) instructions. If decimal mode is set, then the add and subtract instructions work in a different way. The results are left in BCD (binary coded decimal).

Decimal mode off.

3 9 (base sixteen - HEX)

0011 1001 = 57 in decimal.

Decimal mode on.

5 7 (base ten - decimal)

0101 0111 = 57 in decimal

As can be seen, all that happens is that the result is transformed to leave the first decimal digit in the first four bits, and the second, the upper four bits. This is useful for decimal output, rather than binary or hex.

I - Interrupts disabled flag.

This is set if the SEI (set interrupt disabled flag) is executed, and reset if the CLI (clear interrupt disabled flag) instruction is executed. It would not be worthwhile covering this at the moment.

Z - Zero flag.

This is set when any instruction leaves a register with 0 in it. It cannot be set or reset directly. It can be detected with the BEQ (branch if equal), and BNE (branch if not equal) instructions.

C - Carry flag.

This can be thought of as a ninth bit to any register. It can be manipulated directly with the SEC (set carry) and the CLC (clear carry) instructions. It can be detected by the BCC (branch if carry clear) and BCS (branch if carry set) instructions. If 2 two byte (16 bit) values are to be added, and stored in a result, then it is necessary to detect a carry from the first addition of the two low bytes, which should be added to the result of the addition of the two high bytes. An example of this would be as follows:-

Imagine we have two 16 bit values stored in the locations FRED, and BERT. This value is of course split into two 8 bit values. So we have the first value in FRED and FRED+1, and the second in BERT and BERT+1. To add these two values and store the result in JOE and JOE+1, the following code is necessary:-

```
CLC          ;CLEAR CARRY FLAG
             ;START WITH
LDA FRED     ;LOAD THE ACCUMULATOR
             ;WITH THE CONTENTS OF FRED
             ;THIS IS OF COURSE ONLY
             ;THE FIRST 8 BITS OF THE
             ;NUMBER
ADC BERT     ;ADD WITH CARRY THE
             ;CONTENTS OF BERT
STA JOE      ;AND STORE THE RESULT
             ;IN JOE
LDA FRED+1   ;LOAD THE UPPER 8 BITS
             ;OF FRED
ADC BERT+1   ;ADD WITH CARRY THE
             ;UPPER 8 BITS OF BERT
STA JOE+1    ;AND STORE THE RESULT IN
             ;THE UPPER 8 BITS OF JOE
```

At this point it becomes necessary to define what is available to us within the 6502 microprocessor, and its storage. The 6502 uses 2 bytes for each address apart from those in the range $0000 to $00FF (256) bytes, which do not need a high byte to describe their address, because in that range it is always zero. As was stated in the last article, each block of 256 bytes is known as a page of memory. The first 256 bytes ($0000 - $00FF) are known as page zero locations and there is a mode of instruction called simply 'zero page'. This mode tells the processor that it only need pick up one byte for the address.

The 6502 also has what is known as a 'stack', which is used by the processor as well as the programmer. This uses the whole of page one memory (that is $0100 - $01FF). I will try to explain simply what the stack is used for. When doing a GOSUB instruction in BASIC, this tells the PET to jump to a given line number and when a

RETURN instruction is found it must return to the statement following the GOSUB instruction. So, at the time the GOSUB instruction was encountered, it must have some way of remembering where it currently is in memory, so that it knows where to RETURN to. There are instructions PHA (push accumulator on stack), PHP (push processor status register on the stack), PLA (pull accumulator off stack), and PLP (pull processor status off stack), which work with an eight bit register inside the processor called a stack pointer. Each time a PHA instruction is done, the contents of the accumulator will be put at $0100 offset by the stack pointer, and the stack pointer will be automatically deceremented by one, so that the next push will put the data onto the next position on the stack etc. Correspondingly, the PLA instruction will load the accumulator with the contents of the memory whose address is $0100 offset by the stack pointer, and the stack pointer will be incremented by one etc.

N.B The movements of the stack pointer also apply with the PHP and PLP commands. Going back to our GOSUB instruction, by using the instructions just described, the PET may use the stack to save its current position when going off to the subroutine, and when a RETURN instruction is encountered, it simply 'pulls' off the stack, the point at which it was before entering the subroutine, and move onto the next instruction as if nothing had happened. This explains why there is a maximum level of 'nesting' of subroutines (i.e one subroutine calling another, which in turn calls another etc.), because the stack is of a fixed size and as each bunch of return data is pushed onto the stack, the stack slowly gets filled up, until an ?OUT OF MEMORY ERROR occurs. You can demonstrate this to yourself by typing in the BASIC program

The PET should (after a little while) display ?OUT OF MEMORY ERROR IN 10. If you print the value of C, this will tell you the number of levels of subroutine you had got up to before the stack was filled. The PET does in fact check to see if the stack will be over filled by the next instruction. If it didn't, then the PET would probably hang itself up. Fortunately MICROSOFT (the authors of the BASIC interpreter in the PET) thought of that. Right then, we've discussed the stack, and how useful that is, I think I should explain the BRANCH instructions fully now, and I will leave another program for you to look at.

BEQ - Branch if equal, meaning branch if zero flag set.
BNE - Branch if not equal, meaning branch if zero flag not set.
BCC - Branch if carry clear.
BCS - Branch if carry set.
BVC - Branch if overflow clear.
BVS - Branch if overflow set.
BMI - Branch if negative result, branch if negative flag is set
BPL - Branch if positive result, branch if negative flag is not set

PROGRAM TO ROTATE THE SCREEN LEFT BY ONE BYTE

```
*=$033A         ;PROGRAM STARTS AT $033A
                ;
 LDA #$00       ;LOAD ACCUMULATOR WITH
                ;THE VALUE '$00'.
 STA $01        ;STORE IT IN MEMORY
                ;LOCATION $01
 LDA #$80       ;LOAD ACCUMULATOR WITH
                ;THE VALUE '$80'.
 STA $02        ;STORE IT IN MEMORY
                ;LOCATION $02
```

(HERE I HAVE SET UP IN LOCATIONS 1 AND 2 THE 16 BIT ADDRESS OF THE START OF THE SCREEN - $8000. BYTES 1 AND 2 WILL BE USED AS A 16 BIT POINTER TO THE CURRENT LOCATION ON THE SCREEN BEING MOVED)

```
        LDA #$19        ;LOAD ACCUMULATOR WITH
                        ;THE VALUE 25 (DECIMAL)
        STA $00         ;STORE IT IN MEMORY
                        ;LOCATION $00
```

(HERE I HAVE SET UP IN LOCATION 0 A SCREEN LINE COUNTER. I HAVE SET IT TO 25 AND EACH TIME I ROTATE A ROW OF THE SCREEN, I WILL DECREMENT THIS COUNTER, AND STOP WHEN IT REACHES ZERO)

```
START   LDY #0          ;ZEROISE OFFSET TO POINTER
                        ;IN 1 AND 2
        LDA ($01),Y     ;LOAD THE ACCUMULATOR WITH
                        ;THE CONTENTS OF THE ADDRESS
                        ;POINTED TO BY THE BYTES $01
                        ;AND THE NEXT ONE ($02)
                        ;OFFSET BY THE CONTENTS OF THE
                        ;Y REGISTER
        PHA             ;PUSH THE CONTENTS OF THE
                        ;ACCUMULATOR ONTO THE STACK.
                        ;SINCE THIS IS THE FIRST CHARACTER
                        ;IN THE ROW, WE WILL SAVE IT. THEN
                        ;PULL THE OTHER 39 CHARACTERS BACK
                        ;ONE POSITION, AND PULL THIS VALUE
                        ;BACK OFF THE STACK, AND PUT IT AT
                        ;THE END OF THE ROW
        INY             ;INCREMENT THE OFFSET BY ONE
LOOP    LDA ($01),Y     ;GET A CHARACTER FROM THE ROW
        DEY             ;DECREMENT OFFSET TO STORE THIS
                        ;CHARACTER IN THE PREVIOUS POSITION
        STA ($01),Y     ;PUT IT BACK ON THE SCREEN
        INY             ;INCREMENT Y TO WHAT IT WAS
        INY             ;MOVE ONTO THE NEXT SQUARE
        CPY #$28        ;HAVE WE REACHED THE LAST POSITION
                        ;ON THE LINE ? I.E HAS THE OFFSET
                        ;GOT TO 40 (DECIMAL)
        BNE LOOP        ;IF NOT - GO BACK AND DO IT AGAIN
        PLA             ;WE HAVE - SO PUT SAVED CHARACTER
                        ;BACK ONTO THE SCREEN
                        ;PULL BYTE OFF STACK INTO THE
                        ;ACCUMULATOR
        DEY             ;RESET OFFSET TO LAST POSITION
        STA ($01),Y     ;PUT IT IN THE LAST POSITION
        DEC $00         ;DECREMENT LINE COUNT
        BEQ EXIT        ;IF EQUAL TO ZERO - THEN EXIT
        CLC             ;CLEAR CARRY
        LDA $01         ;OTHERWISE - BUMP POINTER BY 40
        ADC #$28        ;ADD $28 TO LOW BYTE
        STA $01         ;AND STORE RESULT
        LDA $02         ;GET HIGH BYTE
        ADC #$00        ;ADD NOTHING + THE CARRY
        JMP START       ;JUMP BACK TO THE START
                        ;
EXIT    RTS             ;FINISHED - RETURN FROM SUBROUTINE
```

INTERRUPT STRUCTURE

Interrupts (including Break or Software Interrupts) are handled by software polling.

When the processor recognises an interrupt it vectors through FFFE and FFFF in ROM to a routine that first inspects the processor hardware (IRQ line low).

If the interrupt was caused by a Break instruction, a Jump indirect is executed through locations $0092 and $0093 (BASIC1: $021B $021C). If it is a hardware interrupt then a Jump Indirect is taken through locations $0090 and $0091 (BASIC1: $0219 $021A).

These locations being in RAM may be user modified to point to extra user code ahead of the normal interrupt processing.

Note, however that the IRQ pointer is used by the cassette routines and should be restored to standard values before cassette Save or Load functions are called.
Various sections of the I/O chips can be set up to cause interrupts through the IRQ line.

Example: POKE 59470,2 enables a negative edge on the user port CAL line to cause an interrupt.

However, your code must be set up for when this happens!

Also note that each pass through the regular interrupt code increments PETs internal clock.

ASSEMBLING AN ASSEMBLER

R.J. Leman, C.Eng. MIERE

Most owners of PETs will, at some time or other, consider the use of machine code for solving problems, particularly if speed of operation is of importance.

Just how long it takes to produce a program depends on the ability of the programmer and the tools, both hardware and software, that he has available.

To show the speed advantage of machine code consider the following example. PET is controlling an experiment and periodically produces five results in the range 5 to 35 from an instrument and logs then into memory. So that the user can keep an eye on progress a simple horizontal bar chart is required, but PET must be "off line" to the experiment for as short a time as possible. This shows a BASIC solution to the problem, and sets up a machine code equivalent in the second cassette buffer.

```
10 S=32768: T=TI: A=40
20 FOR X=5 TO 0 STEP-1
30 A=A-5
40 FORY= 1 TO A
50 POKES+Y,102
60 NEXTY
70 S=S+40
75 PRINT"";PEEK(870)
80 NEXTX: PRINT TI-T "JIFFIES"
200 AD=826
210 READ Q$
220 IF Q$="*" THEN STOP
230 POKEAD,VAL(Q$)
240 AD=AD+1
250 GOTO210
300 DATA169,0,141,76,3,169,128,141,77,3
    ,160,5,190,102,3,169,102,157,200,12
    8,202
310 DATA208,250,136,48,17,173,76,3,24,1
    05,40,141,76,3,144,231,238,77,3,76,
    70,3
320 DATA96,10,15,20,25,30,35,*
```

Type RUN and see the BASIC program draw the bar chart. Now clear the screen, move the cursor down several places and type SYS826. The speed advantage of machine code is quite apparent.

The ability to write machine code is a skill that all PET programmers should acquire, particularly as the SYS and USR commands allow machine code to be mixed with BASIC to produce optimum results. Getting started in BASIC is not too difficult, particularly with the wealth of good books on the subject, but making the first steps in machine code can result in a rapid membership application to Alcoholics Anonymous! The lack of friendly ERROR messages and frustrating hang ups can daunt the most enthusiastic programmer. Obviously a good Assembler package helps with progress, and "hereby hangs the tale".

Assemblers vary in cost, complexity and convenience of use. Particularly inconvenient to the beginner are Assembly Systems requiring repeated loading of Editor, Assembler, debugging package and the necessary files. As an illustration, the time taken to produce the machine code for the example, using a multi-program system, including the correction of a mistyped mnemonic found at the Assembly stage was 56 minutes, of which 30 were spent watching the cassettes go round!!

This frustrating delay resulted in the production of ASMPAC8 and it's related aids.

In a nutshell, ASMPAC8 had to run in my antique 8K PET without any additional memory or provide adequate facilities for entering and editing Assembly Language Text, Assembling into memory or object code file and disassemble programs located in free memory. This was to be done without the use of files other than as a precaution against crash situations

or for record purposes. Obviously a limit on the number of Assembly Language statements had to be accepted, but the ability to produce free standing Object-code in a form suitable for direct saving was essential. Reliability had to be high, the system easy to use and had to adhere to as many of the standard Assembly conventions detailed in the MOS 6502 programming Manual, as possible.

After much development ASMPAC8 was produced and satisfied all of the above requirements. Fifty lines of Assembly Text could be handled without the use of files, and larger requirements dealt with in sections or by the addition of more memory.

Two short programs were produced to operate on the object code files produced by ASMPAC8. AUTOSTRINGS was produced to allow short routines to be stored as coded BASIC strings, and this satisfied the need for a convenient way to include short routines in predominantly BASIC programs.

The obvious place for such code is in the second cassette buffer, but optionally the top 256 bytes may be reserved and used instead. AUTOSTRINGS does this automatically using information from the Object code file.

AUTOLOAD8 accepts Object code files from ASMPAC8 and produces a free standing program that is locatable in the first 2000 bytes of the BASIC RAM area. Such programs may be SAVEd, VERIFIED, LOADed and RUN, just like BASIC but with the enhanced speed that machine code offers.

Both support programs automatically discard their file reading sections once they have finished with them, leaving a tidy program.

Once the programs were in a reasonably polished form, they were tested by the simple technique of use, more use and yet more use. Minor additions such as the ability to display the Label table and check free memory were incorporated. ASMPAC8 is now ready, having met all of its target requirements, to help others learn how to produce and take advantage of the machine code capabilities of PET. One word of warning - it's ADDICTIVE.

ASMPAC8 and its sister programs are approved by Commodore - full details from:
JCL Software, 47 London Road, Southborough, TUNBRIDGE WELLS, Kent.

DIMP

Danny Doyle

DIMP: A machine language routine for the PET to handle algebraic input.

Listing 1 shows a small machine code program called DIMP (short for Direct Input Mode for Pet) which will enable both BASIC and machine language programs to process algebraic expressions. Note that the DIMP routine has been coded to run on the New ROMs.

Strangers to machine language programming will find listing 2 helpful. It shows a short BASIC program which will load DIMP into the second cassette buffer. Once loaded, DIMP can be accessed by SYS826. Listing 3 gives a simple example on how to use DIMP.

For those of you who understand 6502 machine code turn again to listing 1. As you can see the DIMP routine has only 16 instructions, so it can fit easily into the 2nd cassette buffer, or it can be relocated to any convenient memory space without modifications. For certain applications the code can be reduced to only 8 instructions - but more about that later. For the moment let us study listing 1 a little more closely and see what DIMP does. Put simply, when DIMP is called it performs much the same job as the BASIC interpreter when servicing a direct mode command from the keyboard. To see how this is done we will take a walk through the code. Let us assume that DIMP, as per the listing, has been loaded at the beginning of the 2nd cassette buffer at address $033A. Somewhere in his BASIC program the user has coded a SYS826 command at the point where he wishes to input and execute an expression. When a SYS 826 command is executed control is passed to DIMP and this signals the start of phase 1.

The first thing that DIMP does (lines 1-4 in the listing) is to preserve an important pointer; a pointer which contains the address at which BASIC will resume scanning when DIMP has finished processing. DIMP then makes a call (line 5 of the listing) to a firmware routine which asks for a line of input from the keyboard and places the resulting code into the PET keyboard input buffer which is located at address $0200. On return from the keyboard input routine the X and Y registers point to the start address (minus one) of the keyboard input buffer. Line 6 and 7 of the listing shows DIMP storing the X and Y register contents into the location in which they will be used by the interpreter's main scan routine in the next phase of processing. Phase 2 occurs in lines 8 & 9. Line 8 is a call to the scan routine and line 9 calles a ROM routine which converts any keywords in the input image into tokens. (All BASIC statements input into the PET are converted into a machine internal format made up of tokens. Tokens are simply a single byte representation of BASIC keywords such as:- READ, POKE, INPUT, DIM etc.). With the input image converted, DIMP enters its third and main phase.

After a second call to the scan routine (line 10) to reset the image pointer, the main task of evaluating the expression is

performed (line 11) by a call to BASIC's statement execution routine. Note that if any syntax errors are present in the expression the BASIC program will abort and only a cold start using the RUN command will be possible. Assuming that the expression has been sucessfully evaluated, control is returned to DIMP at line 12. This marks the fourth and final phase of processing. All that remains to be done is to restore the statement pointer that was saved on entry and then return control to the BASIC program (lines 12-16).

Until now I have confined my description of DIMP to its ability to handle algebraic expressions entered at run-time. In fact, as well as handling algebraic expressions DIMP will also perform the following subset of BASIC commands:-
DIM; END; PEEK; POKE; PRINT; SAVE; STOP; SYS; WAIT; VERIFY.
Further, with some restrictions, DIMP will handle FOR...NEXT constructions such as:-
FOR I=1 TO N: PRINT SQR(N): NEXT
At the end of the loop the program will halt with a 'READY' message but this can be warm started by typing 'CONT'.

If you are interested in experimenting further with DIMP then here are a few things you might like to think about. Firstly, as shown in listing 1, DIMP has been coded to accept input from the keyboard; this need not be so. By replacing the call to the keyboard input routine in line 5 with a call to a routine of your own design some interesting things can be done. For instance, a routine could be developed which would build an image into the keyboard input buffer, or some other area of memory and with the X and Y registers set appropriately it would then be possible to process a dynamically generated image. Alternatively, one could have 'canned' BASIC images within a machine language program. Then by setting up the X and Y registers with the address (minus one) of a particular image it would be processed by making a call to DIMP with line 5 deleted. Finally, I mentioned earlier that for some applications it would be possible to reduce the size of DIMP. Well if you are going to call DIMP from a machine language program only, you can delete lines 1-4 and 12-15. As already explained these lines handle the saving and restoring of the BASIC statement pointer, so they are not needed for machine language programs.

listing 1

Line	Address	Code	Instruction
1	033A	A5 77	LDA $77
2	033C	48	PHA
3	033D	A5 78	LDA $78
4	033F	48	PHA.
5	0340	20 6F C4	JSR $C46F
6	0343	86 77	STX $77
7	0345	84 78	STY $78
8	0347	20 70 00	JSR $0070
9	034A	20 95 C4	JSR $C495
10	034D	20 70 00	JSR $0070
11	0350	20 00 C7	JSR $C700
12	0353	68	PLA
13	0354	85 78	STA $78
14	0356	68	PLA
15	0357	85 77	STA $77
16	0359	60	RTS

PROGRAM NAME: DIMP PROG

```
10 DATA 165,119,072,165,120,072,032,111

20 DATA 196,134,119,132,120,032,112,000

30 DATA 032,149,196,032,112,000,032,000

40 DATA 199,104,133,120,104,133,119,096

50 REM  ------------------------------
60 FOR A=826 TO 857
70 READ N
80 POKE A,N
90 NEXT
100 PRINT:PRINT"DIMP IS LOADED"
110 NEW
```

PROGRAM NAME: DIMP PROG2

```
10 PRINT"YES? ";
20 SYS826
30 IFA$<>"END"GOTO10
40 STOP
```

[DIMP is a solution looking for a cause, I have so far thought of two areas where it would be very useful but I would like to see what you, the readers can come up with. I am going to give 10 pounds worth of software to any applictions of DIMP which get published in CPUCN. Ed]

LIFE FOR YOUR PET

Dr. F. H. Covitz
New Jersey. USA

Since this is the first time I have attempted to set down a machine language program for the public eye, I will attempt to be as complete as practical without overdoing it.

The programs I will document here are concerned with game of "LIFE", and are written in 6502 machine language specifically for the PET 2001 (8K version) but it will run on any PET. The principles apply to any 6502 system with graphic display capability, and can be debugged (as I did) on non-graphic systems such as the KIM-1.

The first I heard of LIFE was in Martin

Gardner's "Recreational Mathematics" section in Scientific American, Oct-Nov 1970; Feb. 1971. As I understand it, the game was invented by John H. Conway, an English mathematician. In brief, LIFE is a "cellular automation" scheme, where the arena is a rectangular grid (ideally of infinite size). Each square in the grid is either occupied or unoccupied with "seeds", the fate of which are governed by relatively simple rules, i.e. the "facts of LIFE". The rules are 1. A seed survives to the next generation if and only if it has two or three neighbours (right, left, up, down and the four diagonally adjacent cells) otherwise it dies of loneliness or overcrowding, as the case may be. 2. A seed is born in a vacant cell on the next generation if it has exactly 3 neighbours.

With these simple rules, a surprisingly rich game results. The original Scientific American article, and several subsequent articles reveal many curious and surprising initial patterns and results. I understand that there even has been formed a LIFE group, complete with newsletter, although I have not personally seen it.

The game can of course be played manually on a piece of graph paper, but it is slow and prone to mistakes, which have usually disasterous effects on the final results. It would seem to be the ideal thing to put to a microprocessor with bare-bones graphics, since the rules are so simple and there are essentially no arithmetic operaions involved, except for keeping track of addresses and locating neighbours.

As you know, the PET-2001 has an excellent BASIC interpreter, but as yet very little documentation on machine language operation. My first stab was to write a BASIC program, using the entire PET display as the arena (more about that later), and the filled circle graphic display character as the seed. This worked just fine, except for one thing - it took about 2-1/2 minutes for the interpreter to go through one generation! I suppose I shouldn't have been surprised since the program has to check eight neighbouring cells to determine the fate of a particular cell, and do this 1000 times to complete the entire generation (40 x 25 characters for the PET display).

The program following is a 6502 version of LIFE written for the PET. It needs to be POKE'd into the PET memory. I did it with a simple BASIC program and many DATA statements (taking up much more of the program memory space than the actual machine language program!).

The program is accessed by the SYS command, and takes advantage of the display monitor (cursor control) for inserting seeds, and clearing the arena. Without a serious attempt at maximising for speed, the program takes about 1/2 second to go through an entire generation, aout 300 times faster than the BASIC equivalent! Enough said about the efficiency of machine language programming versus BASIC interpreters?

BASIC is great for number crunching, where you can quickly compose your program and have plenty of time to await the results.

The program may be broken down into manageable chunks by subroutining. There follows a brief description of the salient features of each section:

MAIN (hex 1900)

In a fit of overcaution (since this was the first time I attempted to write a PET machine language program) you will notice the series of pushes at the beginning and pulls at the end. I decided to save all the internal registers on the stack in page 1, and also included the CLD (clear decimal mode) just in case. Then follows a series of subroutine calls to do the LIFE generation and display transfers. The zero page location, TIMES, is a counter to permit several loops through LIFE before returning. As set up, TIMES is initialised to zero (hex location 1953) so that it will loop 256 times before jumping back. This of course can be changed either initially or while in BASIC via the POKE command. The return via the JMP BASIC (4C 8B C3) may not be strictly orthodox, but it seems to work all right.

INIT (hex 1930) and DATA (hex 193B)

This shorty reads in the constants needed, and stores them in page zero. SCR refers to the PET screen, TEMP is a temporary working area to hold the new generation as it is evolved, and RCS is essentially a copy of the PET screen data, which I found to be necessary to avoid "snow" on the screen during read/write operations directly on the screen locations. Up, down, etc. are the offsets to be added or subtracted from an address to get all the neighbour addresses. The observant reader will note the gap in the addresses between some of the routines.

TMPSCR (hex 1970)

This subroutine quickly transfers the contents of Temp and dumps it to the screen, using a dot (80 dec) symbol for a live cell (a 1 in TEMP) and a space (32 dec) for the absence of a live cell (a 0 in TEMP).

SCRIMP (hex 198a)

This is the inverse of TMPSCR, quickly transferring (and encoding) data from the screen into TEMP.

RSTORE (hex 19A6)

This subroutine fetches the initial

addresses (high and low) for the SCR, TEMP and RCS memory spaces.

NXTADR (hex 19BD)

Since we are dealing with 1000 bytes of data, we need a routine to increment to the next location, check for page crossing (adding 1 to the high address when it occurs), and checking for the end. The end is signalled by returning a 01 in the accumulator, otherwise a 00 is returned via the accumulator.

TMPRCS (hex 19E6)

The RCS address space is a copy of the screen, used as mentioned before to avoid constant "snow" on the screen if the screen were being continually accessed. This subroutine dumps data from TEMP, where the new generation has been computed, to RCS.

GENER (hex 1A00)

We finally arrive at a subroutine where LIFE is actually generated. After finding out the number of neighbours of the current RCS data byte from NBRS, GENER checks for births (CMPIM $03 at hex addr. 1AOE) if the cell was previously unoccupied. If a birth does not occur, there is an immediate branch to GENADR (the data byte remains 00). If the cell was occupied (CMPIM 81 dec at hex 1AO8), OCC checks for survival (CMPIM $03 at hex 1A1A and CMPIM $02 at hex 1A1E), branching to GENADR when these two conditions are met, otherwise the cell dies (LDAIM $00 at hex 1A22). The results are stored in TEMP for the 1000 cells.

NBRS (hex 1A2F)

NBRS is the subroutine that really does most of the work and where most of the speed could be gained by more efficient programming. Its job, to find the total number of occupied neighbours of a given RCS data location, is complicated by page crossing and edge boundaries. In the present version, page crossing is taken care of, but edge boundaries (left, right, top and bottom of the screen) are somewhat "strange". Above the top line and below the bottom line are considered as sort of forbidden regions where there should practically always be no "life" (data in those regions are not defined by the program, but I have found that there has never been a case where 81's have been present (all other data is considered as "unoccupied" characters). The right and left edges are different, however, and lead to a special type of "geometry". A cell at either edge is not considered as special by NBRS, and so to the right of a right-edge location is the next sequential address. On the screen this is really the left edge location, and one line lower. The inverse is true, of course for left addreses of left-edge locations. Topologically, this is equivalent to a "helix". No special effects of this are seen during a simple LIFE evolution since it just gives the impression of disappearing off one edge while appearing on the other edge. For an object like the "spaceship" (see Scientific American articles), then, the path eventually would cover the whole LIFE arena. The fun comes in when a configuration spreads out so much that it spills over both edges, and interacts with itself. This, of course cannot happen in an infinite universe, so that some of the more complex patterns will not have the same fate in the present version of LIFE. Most of the "blinkers", including the "glider gun" come out OK.

This 40 x 25 version of LIFE can undoubtedly be made more efficient, and other edge algorithms could be found, but I chose to leave it in its original form as a benchmark for my first successfully executed program in writing machine language on the PET. One confession, however - I used the KIM-1 to debug most of the subroutines. Almost all of them did not run on the first shot! Without a good understanding of PET memory allocation particularly in page zero, I was bound to crash many times over, with no recovery other than pulling the plug. The actual BASIC program consisted of POKING loop with many DATA statements (always save on tape before running!).

```
 1 POKE59468,12:POKE537,136:PRINT"■JOHN
   CONWAY'S GAME OF  ▒   ▒ ▒▒▒ ▒▒▒▒"
 2 PRINTSPC(23)"▒   ▒ ▒   ▒":PRINTSPC(23
   )"▒   ▒ ▒▒▒  ▒▒▒":PRINTSPC(23)"▒   ▒ ▒
     ▒"
 3 PRINTSPC(23)"▒▒▒ ▒ ▒   ▒▒▒":PRINT"RUL
   ES:":PRINT"      IF A LIVE CELL IS SU
   RROUNDED BY"
 4 PRINT"TWO OR THREE LIVE CELLS IN THE
   PRESENT  GENERATION IT WILL REMAIN LI
   VE";
 5 PRINT" IN THE   NEXT GENERATION.":
   PRINT"■■■■■■■IF AN EMPTY CELL IS SURR
   OUNDED IN ";
 6 PRINT"THE PRESENT GENERATION BY EXACT
   LY THREE NEIGHBOURS THE CELL WILL BE
   BORN";
 7 PRINT" IN THE NEXT GENERATION.":
   PRINT"■■■■■■■IF A CELL HAS <2 NEIGHBO
   URING";
 8 PRINT" LIVECELLS IT DIES OF LONELINES
   S AND WILL NOTBE PRESENT IN THE NEXT"
   ;
 9 PRINT" GENERATION.":PRINT"■■■■■■■IF A
    CELL HAS FOUR OR MORE LIVE   NEIGHBO
   URS IT";
10 PRINT" WILL DIE FROM OVERCROWDINGIN
   THE NEXT GENERATION. ■■■■■■■■■■■■■■■
   "
11 GETPK$:IFPK$=""THEN11
12 PRINT"■■■A LIVE CELL IS SHOWN BY A S
   HIFTED Q '●'"
13 PRINT"AND A DEAD CELL IS SHOWN AS A
   BLANK ' '"
14 PRINT"TO START GAME DESIGN A PATTERN
    OF":PRINT"SHIFTED Q'S ON THE SCREEN
   "
15 PRINT"AND THEN TYPE 'SYS6400' AND PR
   ESS RETURN":PRINT" e.g.":print
   SPC(15)"●"
```

```
16 PRINTSPC(14)"●●●":PRINTSPC(14)"● ●●■
   SYS6400":PRINT"if you need the inst
   RUCTIONS AGAIN"
17 PRINT"PRESS RETURN OR TO START GAME
   PRESS      ANY OTHER KEY. ?";
19 POKE548,0:GETPK$:IFPK$=""THENPOKE548
   ,1:GOTO19
20 POKE548,1:IFPK$=CHR$(13)THEN1
21 PRINT"■■■■■■■■■"SPC(9)"LOADING MACHIN
   E CODE▶lease wait"

100 READL
110 READA$:C=LEN(A$):IFA$="*"THEN210
120 IFC<10RC>2THEN200
130 A=ASC(A$)-48:B=ASC(RIGHT$(A$,1))-48

140 N=B+7*(B>9)-(C=2)*(16*(A+7*(A>9)))
150 IFN<0ORN>255THEN200
160 POKE59409,52:POKEL,N:POKE59409,60:L
    =L+1:GOTO110
200 PRINT"BYTE"L"=["A$"] ???"
210 PRINT"■■GAME STARTS NOW.":PRINT"CLE
    AR THE SCREEN AND ENTER PATTERN"
220 PRINT"FOLLOWED BY- 'SYS6400'.":FORI
    =1TO4000:NEXT:PRINT"■":POKE537,133:
    END
300 DATA6400
310 DATA20,30,19,20,8A,19,20,E6,19,20,0
    0,1A,A9,34,8D,11,E8
320 DATA20,70,19,A9,3C,8D,11,E8,A9,FF,C
    D,12,E8,F0,E6,4C,8B,C3,AA,68,28,4C,
    8B,C3
330 DATA EA,EA,EA,EA,EA,EA,EA,A2,19,BD,
    3A,19,95,1F,CA,D0,F8,60,00,80,00,15
    ,00
340 DATA80,00,1B,00,1B,D7,28,01,FE;D8,D
    6,29,27,00,E8,83,00,15,00,00
350 DATAEA,EA,EA,EA,EA,EA,EA,EA,EA,EA,E
    A,EA,EA,EA,EA,EA,EA,EA,EA,EA,EA,EA,
    EA
360 DATAEA,EA,EA,EA,EA,20,A6,19,B1,26,D
    0,06,A9,20,91,20,D0,04,A9,51,91,20,
    20
370 DATA BD,19,F0,ED,20,A6,19,60,20,A6,
    19,B1,20,C9,51,F0,06,A9,00,91,26,F0

380 DATA04,A9,01,91,26,20,BD,19,F0,EB,2
    0,A6,19,60,A9,00,AA,A8,85,20,85,26,
    85
390 DATA39,A5,25,85,21,A5,29,85,27,A5,3
    6,85,3A,60,E6,26,E6,20,E6,39,E8,E4
400 DATA33,F0,0C,E0,00,D0,0E,E6,27,E6,2
    1,E6,3A,D0,06,A5,34,C5,21,F0,03,A9,
    00
410 DATA 60,A9,01,60,EA,EA,EA,EA,EA,EA,
    20,A6,19,B1,26,D0,06,A9,20,91,39,D0

420 DATA04,A9,51,91,39,20,BD,19,F0,ED,2
    0,A6,19,60,20,A6,19,20,2F,1A,B1,39,
    C9
430 DATA51,F0,0C,A5,32,C9,03,D0,14,A9,0
    1,91,26,D0,0E,A5,32,C9,03,F0,08,C9,
    02
440 DATAF0,04,A9,00,91,26,20,BD,19,F0,D
    8,20,A6,19,60,98,48,8A,48,A0,00,84,
    32
450 DATAA2,08,B5,29,10,15,49,FF,85,37,3
    8,A5,39,E5,37,85,22,A5,3A,85,23,B0,
    11
460 DATAC6,23,D0,0D,18,65,39,85,22,A5,3
    A,85,23,90,02,E6,23,B1,22,C9,51,D0,
    02
470 DATAE6,32,CA,D0,CF,68,AA,68,A8,60,*
```

```
LINE# LOC   CODE        LINE

0001  0000                          ;************************
0002  0000                          ;***** BINARY LIFE *****
0003  0000                          ;************************
0004  0000                          ;
0005  0000                          ;
0006  0000            BASIC  =$C38B   ;RETURN TO BASIC
0007  0000            OFFSET =$002A   ;DATA POINTER
0008  0000            DOT    =$0051 ;SHIFTED 'Q'
0009  0000            BLANK  =$0020   ;SPACE
0010  0000            ZERO   =$0020 ;START OF DATA TABLE
0011  0000            SCRL   =$0020
0012  0000            SCRH   =$0021
0013  0000            CHL    =$0022
0014  0000            CHH    =$0023
0015  0000            SCRLO  =$0024
0016  0000            SCRHO  =$0025
0017  0000            TEMPL  =$0026
0018  0000            TEMPH  =$0027
0019  0000            TEMPLO =$0028
0020  0000            TEMPHO =$0029
0021  0000            UP     =$002A
0022  0000            DOWN   =$002B
0023  0000            RIGHT  =$002C
0024  0000            LEFT   =$002D
0025  0000            UR     =$002E
0026  0000            UL     =$002F
0027  0000            LR     =$0030
0028  0000            LL     =$0031
0029  0000            N      =$0032
0030  0000            SCRLL  =$0033
```

```
0031  0000             SCRLH  =$0034
0032  0000             RCSLO  =$0035
0033  0000             RCSHO  =$0036
0034  0000             TMP    =$0037
0035  0000             TIMES  =$0038
0036  0000             RCSL   =$0039
0037  0000             RCSH   =$003A
0038  0000
0039  0000                    *=$1900
0040  1900  08          MAIN   PHP              ;SAVE EVERYTHING
0041  1901  48                 PHA             ;ON STACK
0042  1902  8A                 TXA
0043  1903  48                 PHA
0044  1904  98                 TYA
0045  1905  48                 PHA
0046  1906             TSA
0047  1906  8A                 TXA
0048  1907  48                 PHA
0049  1908  D8                 CLD
0050  1909  20 28 19           JSR INIT
0051  190C  20 66 19           JSR SCRTMP
0052  190F  20 BC 19    GEN    JSR TMPRCS
0053  1912  20 D6 19           JSR GENER
0054  1915  20 4C 19           JSR TMPSCR
0055  1918  E6 38              INC TIMES       ;REPEAT 255 TIMES
```

LIFE*.....PAGE 0002

```
LINE# LOC   CODE        LINE

0056  191A  D0 F3              BNE GEN         ;BEFORE QUITTING
0057  191C  68                 PLA             ;RESTORE EVERYTHING
0058  191D  AA                 TAX
0059  191E  9A                 TXS
0060  191F  68                 PLA
0061  1920  A8                 TAY
0062  1921  68                 PLA
0063  1922  AA                 TAX
0064  1923  68                 PLA
0065  1924  28                 PLP
0066  1925  4C 8B C3           JMP BASIC       ;RETURN TO BASIC
0067  1928                                    ;
0068  1928                                    ;
0069  1928                                    ;
0070  1928                                    ;MOVE VALUES INTO ZERO PAGE
0071  1928                                    ;
0072  1928  A2 19        INIT   LDX #$19          ;MOVE 25 VALUES
0073  192A  BD 33 19     LOAD   LDA DATA,X
0074  192D  95 20              STA ZERO,X      ;STORE IN ZERO PAGE
0075  192F  CA                 DEX
0076  1930  D0 F8              BNE LOAD
0077  1932  60                 RTS
0078  1933  00          DATA   .BYTE $00,$80,$00,$15,$00
0078  1934  80
0078  1935  00
0078  1936  15
0078  1937  00
0079  1938  80                 .BYTE $80,$00,$1B,$00,$1B
0079  1939  00
0079  193A  1B
0079  193B  00
0079  193C  1B
0080  193D  D7                 .BYTE $D7,$28,$01,$FE,$D8
0080  193E  28
0080  193F  01
0080  1940  FE
0080  1941  D8
0081  1942  D6                 .BYTE $D6,$29,$27,$00,$E8
0081  1943  29
0081  1944  27
```

```
0081  1945  00
0081  1946  E8
0082  1947  83                .BYTE $83,$00,$15,$00,$00
0082  1948  00
0082  1949  15
0082  194A  00
0082  194B  00
0083  194C                                ;
0084  194C                                ;
0085  194C                                ;
0086  194C  20 82 19    TMPSCR JSR RSTORE    ;GET INIT ADDRESS
0087  194F  B1 26       TSLOAD LDA (TEMPL),Y ;FETCH BYTE FROM TEMP
0088  1951  D0 06              BNE TSONE    ;BRANCH IF NOT ZERO
0089  1953  A5 20              LDA BLANK
0090  1955  91 20              STA (SCRL),Y ;PUT ON SCREEN
```



```
LINE# LOC   CODE        LINE

0091  1957  D0 04              BNE TSNEXT
0092  1959  A5 51        TSONE  LDA DOT
0093  195B  91 20              STA (SCRL),Y
0094  195D  20 99 19    TSNEXT JSR NXTADR
0095  1960  F0 ED              BEQ TSLOAD
0096  1962  20 82 19           JSR RSTORE   ;RESTORE INIT ADDRESS
0097  1965  60                 RTS
0098  1966                                ;
0099  1966                                ;
0100  1966                                ;
0101  1966  20 82 19    SCRTMP JSR RSTORE    ;GET INIT ADDRESS
0102  1969  B1 20       STLOAD LDA (SCRL),Y  ;READ DATA FROM SCREEN
0103  196B  C5 51              CMP DOT      ;TEST FOR DOT
0104  196D  F0 06              BEQ STONE    ;BRANCH IF DOT
0105  196F  A9 00              LDA #$00     ;OTHERWISE IT IS BLANK
0106  1971  91 26              STA (TEMPL),Y ;STORE IT
0107  1973  F0 04              BEQ STNEXT   ;UNCOND. BRANCH
0108  1975  A9 01        STONE  LDA #$01      ;A DOT WAS FOUND
0109  1977  91 26              STA (TEMPL),Y ;STORE IT
0110  1979  20 99 19    STNEXT JSR NXTADR    ;FETCH NEXT ADDRESS
0111  197C  F0 EB              BEQ STLOAD
0112  197E  20 82 19           JSR RSTORE   ;RESTORE INIT ADDRESS
0113  1981  60                 RTS
0114  1982                                ;
0115  1982                                ;
0116  1982                                ;
0117  1982  A9 00       RSTORE LDA #$00      ;ZERO A,X,Y
0118  1984  AA                 TAX
0119  1985  A8                 TAY
0120  1986  85 20              STA SCRL
0121  1988  85 26              STA TEMPL
0122  198A  85 39              STA RCSL
0123  198C  A5 25              LDA SCRHO
0124  198E  85 21              STA SCRH
0125  1990  A5 29              LDA TEMPHO
0126  1992  85 27              STA TEMPH
0127  1994  A5 36              LDA RCSHO
0128  1996  85 3A              STA RCSH
0129  1998  60                 RTS
0130  1999                                ;
0131  1999                                ;
0132  1999                                ;
0133  1999  E6 26       NXTADR INC TEMPL     ;GET NEXT LOW ORDER
0134  199B  E6 20              INC SCRL     ;BYTE ADDRESS
0135  199D  E6 39              INC RCSL
0136  199F  E8                 INX
0137  19A0  E4 33              CPX SCRLL    ;IS IT THE LAST?
0138  19A2  F0 0C              BEQ PAGECH   ;IS IT THE LAST PAGE?
0139  19A4  E4 00              CPX $00      ;IS IT A PAGE BOUNDARY?
0140  19A6  D0 0E              BNE NALOAD   ;IF NOT, THEN NOT DONE
```

```
0141  19A8  E6 27           INC TEMPH      ;OTHERWISE ADVANCE
0142  19AA  E6 21           INC SCRH       ;TO NEXT PAGE.
0143  19AC  E6 3A           INC RCSH
0144  19AE  D0 06           BNE NALOAD     ;UNCONDIT. BRANCH
0145  19B0  A5 34     PAGECH LDA SCRLH      ;CHECK FOR LAST PAGE
```



```
LINE# LOC   CODE        LINE

0146  19B2  C5 21           CMP SCRH
0147  19B4  F0 03           BEQ NADONE     ;IF YES, THEN DONE
0148  19B6  A9 00     NALOAD LDA #$00       ;RETURN WITH A=0
0149  19B8  60               RTS
0150  19B9  A9 01     NADONE LDA #$01       ;RETURN WITH A=1
0151  19BB  60               RTS
0152  19BC                                 ;
0153  19BC                                 ;
0154  19BC                                 ;
0155  19BC  20 82 19  TMPRCS JSR RSTORE
0156  19BF  B1 26     TRLOAD LDA (TEMPL),Y  ;FETCH DATA FROM TEMP
0157  19C1  D0 06           BNE TRONE      ;IF NOT ZERO THEN ALIVE
0158  19C3  A5 20           LDA BLANK
0159  19C5  85 39           STA RCSL       ;STORE IN SCREEN COPY
0160  19C7  D0 04           BNE NEWADR     ;THEN GET NEW ADDRESS
0161  19C9  A5 51     TRONE  LDA DOT
0162  19CB  85 39           STA RCSL       ;STORE IN SCREEN COPY
0163  19CD  20 99 19  NEWADR JSR NXTADR     ;FETCH NEXT ADDRESS
0164  19D0  F0 ED           BEQ TRLOAD     ;IF A=0 THEN NOT DONE
0165  19D2  20 82 19           JSR RSTORE
0166  19D5  60               RTS
0167  19D6                                 ;
0168  19D6                                 ;
0169  19D6                                 ;
0170  19D6  20 82 19  GENER  JSR RSTORE     ;INIT ADDRESSES
0171  19D9  20 05 1A  AGAIN  JSR NBRS       ;FETCH NUMBER OF NEIGHBORS
0172  19DC  A5 39           LDA RCSL       ;FETCH CURRENT DATA
0173  19DE  C5 51           CMP DOT        ;IS IT A DOT?
0174  19E0  F0 0C           BEQ OCC        ;IF YES THEN BRANCH
0175  19E2  A5 32           LDA N          ;OTHERWISE ITS BLANK
0176  19E4  C9 03           CMP #$03       ;SO WE CHECK
0177  19E6  D0 14           BNE GENADR     ;FOR A BIRTH
0178  19E8  A9 01     BIRTH  LDA #$01       ;IT GIVES BIRTH
0179  19EA  85 26           STA TEMPL      ;STORE IN TEMP
0180  19EC  D0 0E           BNE GENADR     ;UNCOND BRANCH
0181  19EE  A5 32     OCC    LDA N          ;FETCH NO. OF NEIGHBORS
0182  19F0  C9 03           CMP #$03       ;IF IT HAS 3
0183  19F2  F0 08           BEQ GENADR     ; OR
0184  19F4  C9 02           CMP #$02       ;2 NEIGHBORS IT
0185  19F6  F0 04           BEQ GENADR     ;SURVIVES
0186  19F8  A9 00     DEATH  LDA #$00       ;IT DIED!
0187  19FA  91 26           STA (TEMPL),Y  ;STORE IN TEMP
0188  19FC  20 99 19  GENADR JSR NXTADR     ;FETCH NEXT ADDRESS
0189  19FF  F0 D8           BEQ AGAIN      ;IF 0, THEN NOT DONE
0190  1A01  20 82 19           JSR RSTORE
0191  1A04  60               RTS
0192  1A05                                 ;
0193  1A05                                 ;
0194  1A05                                 ;
0195  1A05  98         NBRS   TYA            ;SAVE Y & X ON STACK
0196  1A06  48               PHA
0197  1A07  8A               TXA
0198  1A08  48               PHA
0199  1A09  A0 00           LDY #$00 SET Y AND N=0
0200  1A0B  84 32           STY N
```

```
LIFE*.....PAGE 0005

LINE# LOC   CODE        LINE

0201  1A0D  A6 08              LDX #08         ;CHECK 8 NEIGHBORS
0202  1A0F  B5 2A        OFFS  LDA OFFSET,X
0203  1A11  10 15              BPL ADD         ;ADD OFFSET IF POSITIVE
0204  1A13  49 FF              EOR ##FF        ;OTHERWISE SET
0205  1A15  85 37              STA TMP         ;FOR SUBTRACT
0206  1A17  38                 SEC
0207  1A18  A5 39              LDA RCSL
0208  1A1A  E5 37              SBC TMP         ;SUBTRACT TO GET THE
0209  1A1C  85 22              STA CHL         ;CORRECT NEIGHBOR ADDRESS
0210  1A1E  A5 3A              LDA RCSH
0211  1A20  85 23              STA CHH
0212  1A22  B0 11              BCS EXAM        ;OK FIND OUT WHATS THERE
0213  1A24  C6 23              DEC CHH         ;PAGE CROSS
0214  1A26  D0 0D              BNE EXAM        ;UNCOND BRANCH
0215  1A28  18            ADD   CLC              ;SET UP FOR ADD
0216  1A29  65 39              ADC RCSL        ;ADD
0217  1A2B  85 22              STA CHL         ;STORE THE LOW PART
0218  1A2D  A5 3A              LDA RCSH        ;FETCH THE HIGH PART
0219  1A2F  85 23              STA CHH
0220  1A31  90 02              BCC EXAM        ;OK WHATS THERE
0221  1A33  E6 23              INC CHH
0222  1A35  B1 22        EXAM  LDA (CHL),Y      ;FETCH THE NEIGHBOR
0223  1A37  C5 51              CMP DOT         ;DATA BYTE AND SEE IF
0224  1A39  D0 02              BNE NEXT        ;OCCUPIED
0225  1A3B  E6 32              INC N           ;ACCUMULATE THE NUMBER OF NE
S
0226  1A3D  CA            NEXT  DEX
0227  1A3E  D0 CF              BNE OFFS        ;NOT DONE
0228  1A40  68                 PLA             ;RESTORE X Y FROM STACK
0229  1A41  AA                 TAX
0230  1A42  68                 PLA
0231  1A43  A8                 TAY
0232  1A44  60                 RTS
0233  1A45                      END

ERRORS = 0000

SYMBOL TABLE

SYMBOL VALUE
 ADD      1A28    AGAIN    19D9    BASIC    C38B    BIRTH    19E8
 BLANK    0020    CHH      0023    CHL      0022    DATA     1933
 DEATH    19F8    DOT      0051    DOWN     002B    EXAM     1A35
 GEN      190F    GENADR   19FC    GENER    19D6    INIT     1928
 LEFT     002D    LL       0031    LOAD     192A    LR       0030
 MAIN     1900    N        0032    NADONE   19E9    NALOAD   19B6
 NBRS     1A05    NEWADR   19CD    NEXT     1A3D    NXTADR   1939
 OCC      19EE    OFFS     1A0F    OFFSET   002A    PAGECH   19B0
 RCSH     003A    RCSHO    0036    RCSL     0039    RCSLO    0035
 RIGHT    002C    RSTORE   1982    SCRH     0021    SCRHO    0025
 SCRL     0020    SCRLH    0034    SCRLL    0033    SCRLO    0024
 SCRTMP   1966    STLOAD   1969    STNEXT   1979    STONE    1975
 TEMPH    0027    TEMPHO   0029    TEMPL    0026    TEMPLO   0028

SYMBOL TABLE

SYMBOL VALUE
 TIMES    0038    TMP      0037    TMPRCS   19BC    TMPSCR   194C
 TRLOAD   19BF    TRONE    19C9    TSA      1906    TSLOAD   194F
 TSNEXT   195D    TSONE    1959    UL       002F    UP       002A
 UR       002E    ZERO     0020

END OF ASSEMBLY
```

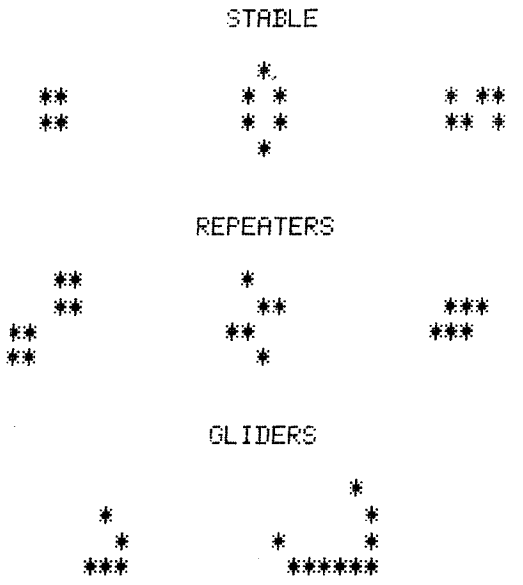
A BRIEF INTRODUCTION TO THE GAME OF LIFE

One of the interesting properties of the game of LIFE is that such simple rules can lead to such complex activity. The simplicity comes from the fact that the rules apply to each individual cell. The complexity comes from the interactions between the individual cells. Each individual cell is affected by its eight adjacent neighbours, and nothing else.

The rules are:

1. A cell survives if it has two or three neighbors.
2. A cell dies from overcrowding if it has four or more neighbors. It dies from isolation if it has one or zero neighbors.
3. A cell is born when an empty space has exactly three neighbors.

With these few rules, many different types of activity can occur. Some patterns are STABLE, that is they do not change at all. Some are REPEATERS, patterns which undergo one or more changes and return to the original pattern. A REPEATER may repeat as fast as every other generation, or may have a longer period. A GLIDER is a pattern which moves as it repeats.



LIFE FOR YOUR PET

Below we have a way of actually getting our HEX OP-CODES into the PET. Lines 100-200 read the data statements convert them to decimal and POKE them sequentially into the memory. The first data item is expected to be the starting point of the loading in decimal and the last data item is expected to be an asterix. The beauty of this method is that you can use the screen edit facility on the PET for inserting and deleting codes. When you have inserted your own data statements from line 300 upwards, save the entire performance prior to running as machine language routines rarely work first time around and the PET is quite likely to hang up and need turning off and on. The data statements in the example are for the game of LIFE. In the original version listed on the previous pages, 256 generations must occur before the control returns to BASIC. I have modified the program slightly in the beginning in order to allow the stop button to halt the binary program. When the machine prints READY, clear the screen. Type say eight shifted Q's in a row in the middle of the screen followed by SYS (6400) (which is 1900H in decimal) and press return. GOOD LUCK!

SUPERMON 2

Supermon is a powerful machine code utility for the PET. It has a number of very useful facilities which will aid in the debugging and writing of machine code programs. The following list shows the instructions which are available. The instructions include those which are already available in TIM the resident monitor.

```
[illegible]
commands - user input in [illegible]
[illegible]
.[illegible]
.A 2002 [illegible]
.A 2005 [illegible]
.
```

IN THE ABOVE EXAMPLE THE USER STARTED ASSEMBLY AT 1000 HEX. THE FIRST INSTRUCTION WAS LOAD A REGISTER WITH IMMEDIATE 12 HEX. IN THE SECOND LINE THE USER DID NOT NEED TO TYPE THE A AND ADDRESS. THE SIMPLE ASSEMBLER PROMPTS WITH THE NEXT ADDRESS. TO EXIT THE ASSEMBLER TYPE A RETURN AFTER THE THE ADDRESS PROMPT. SYNTAX IS THE SAME AS THE DISASSEMBLER OUTPUT.

```
[illegible]
.[illegible]
(SCREEN CLEARS)
.; 2000 A9 12       LDA ##12
.; 2002 9D 00 80    STA $8000,X
.; 2005 AA          TAX
.; 2006 AA          TAX
(FULL PAGE OF INSTRUCTIONS)
```

DISASSEMBLES 22 INSTRUCTIONS STARTING AT 1000 HEX. THE THREE BYTES FOLLOWING THE ADDRESS MAY BE MODIFIED. USE THE CRSR KEYS TO MOVE TO AND MODIFY THE BYTES. HIT RETURN AND THE BYTES IN MEMORY WILL BE CHANGED. [illegible] WILL THEN DISASSEMBLE THAT PAGE AGAIN.

[redacted]
.[redacted]
```
2000 A9 12       LDA #$12
2002 9D 00 80    STA $8000,XY.
2005 AA          TAX
  ....
203F A2 00       LDX #$00
```
TO ENGAGE PRINTER, SET UP BEFOREHAND:
[redacted]
AND ACCESS THE MONITOR VIA A CALL
(*NOT* A BREAK) COMMAND

[redacted]
.[redacted]

ALLOWS A MACHINE LANGUAGE PROGRAM TO BE RUN STEP BY STEP.
CALL REGISTER DISPLAY WITH .[redacted] AND SET THE PC ADDRESS TO THE DESIRED FIRST INSTRUCTION FOR SINGLE STEPPING.
THE .[redacted] WILL CAUSE A SINGLE STEP TO EXECUTE AND WILL DISASSEMBLE THE NEXT.
CONTROLS:
- [redacted] FOR SINGLE STEP;
- [redacted] FOR SLOW STEP;
- [redacted] FOR FAST STEPPING;
- [redacted] TO RETURN TO MONITOR.

[redacted]
.[redacted] [redacted] [redacted] [redacted]

FILLS THE MEMORY FROM 1000 HEX TO 1100 HEX WITH THE BYTE FF HEX.

[redacted]
.[redacted]

GO TO THE ADDRESS IN THE PC REGISTER DISPLAY AND BEGIN RUN CODE.
ALL THE REGISTERS WILL BE REPLACED WITH THE DISPLAYED VALUES.
.[redacted] [redacted]

GO TO ADDRESS 1000 HEX AND BEGIN RUNNING CODE.

[redacted]
.[redacted] [redacted] [redacted] [redacted]

HUNT THRU MEMORY FROM C000 HEX TO D000 HEX FOR THE ASCII STRING [redacted] AND PRINT THE ADDRESS WHERE IT IS FOUND. A MAXIMUM OF 32 CHARACTERS MAY BE USED.
.[redacted] [redacted] [redacted] [redacted] [redacted]

HUNT MEMORY FROM C000 HEX TO D000 HEX FOR THE SEQUENCE OF BYTES 20 D2 FF AND PRINT THE ADDRESS. A MAXIMUM OF 32 BYTES MAY BE USED.

[redacted]
.[redacted]

LOAD ANY PROGRAM FROM CASSETTE #1.
.[redacted] [redacted]

LOAD FROM CASSETTE #1 THE PROGRAM NAMED [redacted].
.[redacted] [redacted]

LOAD FROM CASSETTE #2 THE PROGRAM NAMED [redacted].

[redacted]
.[redacted]

```
     PC  IRQ  SR AC XR YR SP
.;  0000 E62E 01 02 03 04 05
```

DISPLAYS THE REGISTER VALUES SAVED WHEN [redacted] WAS ENTERED. THE VALUES MAY BE CHANGED WITH THE EDIT FOLLOWED BY A RETURN.
USE THIS INSTRUCTION TO SET UP THE PC VALUE BEFORE SINGLE STEPPING WITH .[redacted]

[redacted]
.[redacted] [redacted]

SAVE TO CASSETTE #1 MEMORY FROM 0800 HEX UP TO BUT NOT INCLUDING 0C80 HEX AND NAME IT [redacted].

[redacted]
.[redacted] [redacted] [redacted] [redacted]

TRANSFER MEMORY IN THE RANGE 1000 HEX TO 1100 HEX AND START STORING IT AT ADDRESS 5000 HEX.

[redacted]
.[redacted]

RETURN TO BASIC READY MODE.
THE STACK VALUE SAVED WHEN ENTERED WILL BE RESTORED. CARE SHOULD BE TAKEN THAT THIS VALUE IS THE SAME AS WHEN THE MONITOR WAS ENTERED. A CLR IN BASIC WILL FIX ANY STACK PROBLEMS.

SUMMARY

COMMODORE MONITOR INSTRUCTIONS:

- [redacted] GO RUN
- [redacted] LOAD FROM TAPE
- [redacted] MEMORY DISPLAY
- [redacted] REGISTER DISPLAY
- [redacted] SAVE TO TAPE
- [redacted] EXIT TO BASIC

SUPERMON ADDITIONAL INSTRUCTIONS:

- [redacted] SIMPLE ASSEMBLER
- [redacted] DISASSEMBLER
- [redacted] FILL MEMORY
- [redacted] HUNT MEMORY
- [redacted] SINGLE INSTRUCTION
- [redacted] PRINTING DISASSEMBLER
- [redacted] TRANSFER MEMORY

[redacted] WILL LOAD ITSELF INTO THE TOP OF MEMORY .. WHEREVER THAT HAPPENS TO BE ON YOUR MACHINE.
YOU MAY THEN SAVE THE MACHINE CODE FOR FASTER LOADING IN THE FUTURE.
BE SURE TO NOTE THE SYS COMMAND WHICH LINKS [redacted] TO THE COMMODORE MONITOR.

The following listing gives you a program which will print the above instructions to the screen of the PET.

```
10 REM SUPERMON INSTR
1100 GOSUB10000
1200 PRINT"■■■ SIMPLE ASSEMBLER "
1300 PRINT"■. ■A■  ■2000■  ■LDA■  ■#$12
1310 PRINT".A 2002 ■STA■  ■#8000,X
1320 PRINT".A 2005 ■(RETURN)
1330 PRINT".
1340 PRINT"      IN THE ABOVE EXAMPLE TH
     E USER
1350 PRINT"STARTED ASSEMBLY AT 1000 HEX
     .  THE
1360 PRINT"FIRST INSTRUCTION WAS LOAD A
      REGISTER
1370 PRINT"WITH IMMEDIATE 12 HEX.  IN T
     HE SECOND
1380 PRINT"LINE THE USER DID NOT NEED T
     O TYPE THE
1390 PRINT"A AND ADDRESS.  THE SIMPLE A
     SSEMBLER
1400 PRINT"PROMPTS WITH THE NEXT ADDRES
     S.  TO EXIT
1410 PRINT"THE ASSEMBLER TYPE A RETURN
     AFTER THE
1420 PRINT"THE ADDRESS PROMPT.  SYNTAX
     IS THE SAME
1430 PRINT"AS THE DISASSEMBLER OUTPUT.
1450 GOSUB9000
1500 PRINT"■■■  DISASSEMBLER "
1510 PRINT"■. ■D■  ■2000■
1520 PRINT"■(SCREEN CLEARS)
1530 PRINT".,  2000 A9 12        LDA #$1
     2
1540 PRINT".,  2002 9D 00 80     STA $80
     00,X
1550 PRINT".,  2005 AA           TAX
1560 PRINT".,  2006 AA           TAX
1570 PRINT"(FULL PAGE OF INSTRUCTIONS)
1600 PRINT"■      DISASSEMBLES 22 INSTRU
     CTIONS
1610 PRINT"STARTING AT 1000 HEX.  THE T
     HREE BYTES
1620 PRINT"FOLLOWING THE ADDRESS MAY BE
      MODIFIED.
1630 PRINT"USE THE CRSR KEYS TO MOVE TO
      AND MODIFY
1640 PRINT"THE BYTES.  HIT RETURN AND T
     HE BYTES
1650 PRINT"IN MEMORY WILL BE CHANGED.
     ■■■■■■■■■
1660 PRINT"WILL THEN DISASSEMBLE THAT P
     AGE AGAIN.
1690 GOSUB9000
1700 PRINT"■■■ PRINTING DISASSEMBLER "
1710 PRINT"■. ■P■  ■2000,2040■
1720 PRINT"2000 A9 12        LDA #$12
1730 PRINT"2002 9D 00 80     STA $8000,X
     Y.
1740 PRINT"2005 AA           TAX
1750 PRINT"    ...."
1760 PRINT"203F A2 00        LDX #$00
1770 PRINT"■TO ENGAGE PRINTER, SET UP B
     EFOREHAND:
1775 PRINT"      ■OPEN 4,4:CMD4
1780 PRINT"AND ACCESS THE MONITOR VIA A
      CALL
1785 PRINT"(*NOT* A BREAK) COMMAND
1790 GOSUB 9000
1800 PRINT"■■■ SINGLE STEP "
1810 PRINT"■. ■I
1820 PRINT"■      ALLOWS A MACHINE LANGU
     AGE PROGRAM
1830 PRINT"TO BE RUN STEP BY STEP.
1840 PRINT"■CALL REGISTER DISPLAY WITH
     .■ AND SET
1850 PRINT"THE PC ADDRESS TO THE DESIRE
     D FIRST
1860 PRINT"INSTRUCTION FOR SINGLE STEPP
     ING.
1870 PRINT"THE .■I■ WILL CAUSE A SINGLE
      STEP TO
1880 PRINT"EXECUTE AND WILL DISASSEMBLE
      THE NEXT.
1890 PRINT"■CONTROLS:
1900 PRINT"  ■C■ FOR SINGLE STEP;
1910 PRINT"  ■RVS■ FOR SLOW STEP;
1920 PRINT"  ■SPACE■ FOR FAST STEPPING;
1930 PRINT"  ■STOP■ TO RETURN TO MONITO
     R."
1990 GOSUB9000
2000 PRINT"■■■■ FILL MEMORY "
2010 PRINT"■■. ■F■  ■1000■  ■1100■  ■FF
2020 PRINT"■      FILLS THE MEMORY FROM
     1000 HEX TO
2030 PRINT"1100 HEX WITH THE BYTE FF HE
     X.
2090 GOSUB9000
2100 PRINT"■■■■ GO RUN "
2110 PRINT"■. ■G
2120 PRINT"■      GO TO THE ADDRESS IN T
     HE PC
2130 PRINT"REGISTER DISPLAY AND BEGIN R
     UN CODE.
2140 PRINT"ALL THE REGISTERS WILL BE RE
     PLACED
2150 PRINT"WITH THE DISPLAYED VALUES."
2160 PRINT"■. ■G■  ■1000
2170 PRINT"■      GO TO ADDRESS 1000 HEX
      AND BEGIN
2180 PRINT"RUNNING CODE.
2190 GOSUB9000
2200 PRINT"■■■■ HUNT MEMORY "
2210 PRINT"■. ■H■  ■C000■  ■D000■  ■'READ
2220 PRINT"■      HUNT THRU MEMORY FROM
     C000 HEX TO
2230 PRINT"D000 HEX FOR THE ASCII STRIN
     G ■■■■ AND
2240 PRINT"PRINT THE ADDRESS WHERE IT I
     S FOUND.  A
2250 PRINT"MAXIMUM OF 32 CHARACTERS MAY
      BE USED.
2260 PRINT"■. ■H■  ■C000■  ■D000■  ■20■  ■D2
      ■■
2270 PRINT"■      HUNT MEMORY FROM C000
     HEX TO D000
2280 PRINT"HEX FOR THE SEQUENCE OF BYTE
     S 20 D2 FF
2290 PRINT"AND PRINT THE ADDRESS.  A MA
     XIMUM OF 32
2300 PRINT"BYTES MAY BE USED.
2390 GOSUB9000
2400 PRINT"■■ LOAD FROM TAPE "
2401 PRINT"■■. ■L■
2402 PRINT"■   LOAD ANY PROGRAM FROM CA
     SSETTE #1.
2403 PRINT"■■. ■L■ ■";CHR$(34);"RAM TEST
     ";CHR$(34)
2404 PRINT"■   LOAD FROM CASSETTE #1 TH
     E PROGRAM
2405 PRINT"NAMED ■RAM TEST■.
2410 PRINT"■■. ■L■ ■";CHR$(34);"RAM TEST
     ";CHR$(34);",02
2420 PRINT"■   LOAD FROM CASSETTE #2 TH
     E PROGRAM
2430 PRINT"NAMED ■RAM TEST■.
2490 GOSUB9000
2500 PRINT"■■■ MEMORY DISPLAY "
```

```
2510 PRINT"▮.▮M▮ ▮0000▮ ▮0080
2520 PRINT"▮.:  0000 00 01 02 03 04 05
     06 07
2530 PRINT".:  0008 08 09 0A 0B 0C 0D 0
     E 0F
2540 PRINT"▮      DISPLAY MEMORY FROM 00
     00 HEX TO
2550 PRINT"0080 HEX.  THE BYTES FOLLOWI
     NG THE
2560 PRINT"ADDRESS MAY BE MODIFIED BY E
     DITING AND
2570 PRINT"THEN TYPING A RETURN.
2590 GOSUB9000
2600 PRINT"▮▮▮ REGISTER DISPLAY "
2610 PRINT"▮.▮R
2620 PRINT"▮      PC  IRQ  SR AC XR YR S
     P
2630 PRINT".;  0000 E62E 01 02 03 04 05

2640 PRINT"▮      DISPLAYS THE REGISTER
     VALUES SAVED
2650 PRINT"WHEN ▮SUPERMON▮ WAS ENTERED.
      THE VALUES
2660 PRINT"MAY BE CHANGED WITH THE EDIT
      FOLLOWED
2670 PRINT"BY A RETURN.
2671 PRINT"▮  USE THIS INSTRUCTION TO S
     ET UP THE
2672 PRINT"PC VALUE BEFORE SINGLE STEPP
     ING WITH"
2673 PRINT".▮I▮
2690 GOSUB9000
2700 PRINT"▮▮▮▮ SAVE TO TAPE "
2710 PRINT"▮.▮S▮ ▮";CHR$(34);"PROGRAM N
     AME";CHR$(34);",01,0800,0C80"
2720 PRINT"▮      SAVE TO CASSETTE #1 ME
     MORY FROM
2730 PRINT"0800 HEX UP TO BUT NOT INCLU
     DING 0C80
2740 PRINT"HEX AND NAME IT ▮PROGRAM NAM
     E.
2790 GOSUB9000
2800 PRINT"▮▮▮▮▮ TRANSFER MEMORY "
2810 PRINT"▮.▮T▮ ▮1000▮ ▮1100▮ ▮5000
2820 PRINT"▮      TRANSFER MEMORY IN THE
      RANGE 1000
2830 PRINT"HEX TO 1100 HEX AND START ST
     ORING IT AT
2840 PRINT"ADDRESS 5000 HEX.
2890 GOSUB9000
3200 PRINT"▮▮▮▮ EXIT TO BASIC "
3210 PRINT"▮.▮X
3220 PRINT"▮      RETURN TO BASIC READY
     MODE.
3230 PRINT"THE STACK VALUE SAVED WHEN E
     NTERED WILL
3240 PRINT"BE RESTORED.  CARE SHOULD BE
      TAKEN THAT
3250 PRINT"THIS VALUE IS THE SAME AS WH
     EN THE
3260 PRINT"MONITOR WAS ENTERED.  A CLR
     IN
3270 PRINT"BASIC WILL FIX ANY STACK PRO
     BLEMS.
3290 GOSUB9000
3500 PRINT"▮          SUMMARY

3505 PRINT"COMMODORE MONITOR INSTRUCTIO
     NS:"
3510 PRINT"▮G▮ GO RUN
3520 PRINT"▮L▮ LOAD FROM TAPE
3530 PRINT"▮M▮ MEMORY DISPLAY
3540 PRINT"▮R▮ REGISTER DISPLAY
3550 PRINT"▮S▮ SAVE TO TAPE
3560 PRINT"▮X▮ EXIT TO BASIC
3595 PRINT"▮SUPERMON ADDITIONAL INSTRUC
     TIONS:"
3600 PRINT"▮▮A▮ SIMPLE ASSEMBLER
3610 PRINT"▮D▮ DISASSEMBLER
3620 PRINT"▮F▮ FILL MEMORY
3630 PRINT"▮H▮ HUNT MEMORY
3640 PRINT"▮I▮ SINGLE INSTRUCTION
3650 PRINT"▮P▮ PRINTING DISASSEMBLER
3660 PRINT"▮T▮ TRANSFER MEMORY
4000 GOSUB9000
4010 PRINT"▮▮SUPERMON▮ WILL LOAD ITSELF
      INTO THE
4020 PRINT"TOP OF MEMORY .. WHEREVER TH
     AT HAPPENS
4030 PRINT"TO BE ON YOUR MACHINE.
4040 PRINT"▮  YOU MAY THEN SAVE THE MA
     CHINE CODE
4050 PRINT"FOR FASTER LOADING IN THE FU
     TURE.
4060 PRINT"▮BE SURE TO NOTE THE SYS COM
     MAND WHICH
4070 PRINT"LINKS ▮SUPERMON▮ TO THE COMM
     ODORE
4080 PRINT"MONITOR.
5000 GOSUB9000
8000 GOTO1200
9000 PRINT"▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
      ▮HIT▮ANY▮KEY▮TO▮CONTINUE▮";
9010 FORI=1TO10:GETA$:NEXT
9020 GETA$:IFA$=""THEN9020
10000 PRINT"▮          ▮ S U P E R M O N
       2 . 0 "
10010 PRINT"▮COMMANDS - USER INPUT IN ▮
       REVERSE "
10040 RETURN
```

This next listing allows you to load BASIC2 Supermon into whatever memory you have available. Type in the program and then save it on disk or cassette. This will only work with BASIC2 Supermon. BASIC1 Supermon is limited to the memory locations $16F6-$1FFF.

```
100 PRINT"▮▮▮▮ SUPERMON2!"
110 PRINT"▮      DISSASSEMBLER ▮D▮ BY WO
    ZNIAK/BAUM
120 PRINT"       SINGLE STEP ▮I▮ BY JIM
     RUSSO
130 PRINT"MOST OTHER STUFF ▮,HAFT▮ BY B
    ILL SEILER
150 PRINT"▮TIDIED & WRAPPED BY JIM BUTT
    ERFIELD"
170 L=PEEK(52)+PEEK(53)*256:SYS1536:M=
    PEEK(33):N=PEEK(34)
180 POKE52,M:POKE53,N:POKE48,M:POKE49,N
    :N=M+N*256
210 PRINT"▮▮LINK TO MONITOR -- SYS";N
220 PRINT:PRINT"SAVE WITH MLM:"
230 PRINT".S ";CHR$(34);"SUPERMON";
    CHR$(34);",01";:X=N/4096:GOSUB250
240 X=L/4096:GOSUB250:END
250 PRINT",";:FORJ=1TO4:X%=X:X=(X-X%)*1
    6:IFX%>9THENX%=X%+7
260 PRINTCHR$(X%+48);:NEXTJ:RETURN
```

Now type in the following BASIC2 Supermon machine code using TIM, access TIM with SYS4. Save the program before running it.

.S "1:SUPERMON",08,0600,0D00
This will save SUPERMON on drive 1 of the disk. Remember to initialise the disk before entering the monitor!

.S "SUPERMON",01,0600,0D00
This will save SUPERMON on cassette #1

Machine code is very easy to input wrongly and will more than likely crash your PET, so make sure you have made a copy of the program before running it. Lower the top of memory to protect the program from BASIC by typing:-
POKE 53,6: POKE 52,0 this is for BASIC2.

```
.
.: 0600 A9 CB 85 1F A9 0C 85 20
.: 0608 A5 34 85 21 A5 35 85 22
.: 0610 A0 00 20 38 06 D0 16 20
.: 0618 38 06 F0 11 85 23 20 38
.: 0620 06 18 65 34 AA A5 23 65
.: 0628 35 20 43 06 8A 20 43 06
.: 0630 20 50 06 90 DB 60 EA EA
.: 0638 A5 1F D0 02 C6 20 C6 1F
.: 0640 B1 1F 60 48 A5 21 D0 02
.: 0648 C6 22 C6 21 68 91 21 60
.: 0650 A9 80 C5 1F A9 06 E5 20
.: 0658 60 AA AA AA AA AA AA AA
.: 0660 AA AA AA AA AA AA AA AA
.: 0668 AA AA AA AA AA AA AA AA
.: 0670 AA AA AA AA AA AA AA AA
.: 0678 AA AA AA AA AA AA AA AA
.: 0680 AD FE FF 00 85 34 AD FF
.: 0688 FF 00 85 35 AD FC FF 00
.: 0690 8D FA 03 AD FD FF 00 8D
.: 0698 FB 03 00 00 A2 08 DD DE
.: 06A0 FF 00 D0 0E 86 B4 8A 0A
.: 06A8 AA BD E9 FF 00 48 BD E8
.: 06B0 FF 00 48 60 CA 10 EA 4C
.: 06B8 9A FA 00 A2 02 2C A2 00
.: 06C0 00 B4 FB D0 08 B4 FC D0
.: 06C8 02 E6 DE D6 FC D6 FB 60
.: 06D0 20 EB E7 C9 20 F0 F9 60
.: 06D8 A9 00 00 8D 00 00 01 20
.: 06E0 79 FA 00 20 BE E7 20 AA
.: 06E8 E7 90 09 60 20 EB E7 20
.: 06F0 A7 E7 B0 DE AE 1A 02 9A
.: 06F8 4C F7 E7 20 CD FD CA D0
.: 0700 FA 60 E6 FD D0 02 E6 FE
.: 0708 60 A2 02 B5 FA 48 BD 0A
.: 0710 02 95 FA 68 9D 0A 02 CA
.: 0718 D0 F1 60 AD 0B 02 AC 0C
.: 0720 02 4C CE FA 00 A5 FD A4
.: 0728 FE 38 E5 FB 8D 1B 02 98
.: 0730 E5 FC A8 0D 1B 02 60 20
.: 0738 81 FA 00 20 97 E7 20 92
.: 0740 FA 00 20 AF FA 00 20 92
.: 0748 FA 00 20 CA FA 00 20 97
.: 0750 E7 90 15 A6 DE D0 65 20
.: 0758 C1 FA 00 90 60 A1 FB 81
.: 0760 FD 20 A8 FA 00 20 D5 FD
.: 0768 D0 EB 20 C1 FA 00 18 AD
.: 0770 1B 02 65 FD 85 FD 98 65
.: 0778 FE 85 FE 20 AF FA 00 A6
.: 0780 DE D0 3D A1 FB 81 FD 20
.: 0788 C1 FA 00 B0 34 20 65 FA
.: 0790 00 20 68 FA 00 4C 1B FB
.: 0798 00 20 81 FA 00 20 97 E7
.: 07A0 20 92 FA 00 20 97 E7 20
.: 07A8 EB E7 20 B6 E7 90 14 85
.: 07B0 B5 A6 DE D0 11 20 CA FA
.: 07B8 00 90 0C A5 B5 81 FB 20
.: 07C0 D5 FD D0 EE 4C 9A FA 00
.: 07C8 4C 56 FD 20 81 FA 00 20
.: 07D0 97 E7 20 92 FA 00 20 97
.: 07D8 E7 20 EB E7 A2 00 00 20
.: 07E0 EB E7 C9 27 D0 14 20 EB
.: 07E8 E7 9D 10 02 E8 20 CF FF
.: 07F0 C9 0D F0 22 E0 20 D0 F1
.: 07F8 F0 1C 8E 00 00 01 20 BE
.: 0800 E7 90 C6 9D 10 02 E8 20
.: 0808 CF FF C9 0D F0 09 20 B6
.: 0810 E7 90 B6 E0 20 D0 EC 86
.: 0818 B4 20 D0 FD A2 00 00 A0
.: 0820 00 00 B1 FB DD 10 02 D0
.: 0828 0C C8 E8 E4 B4 D0 F3 20
.: 0830 6A E7 20 CD FD 20 D5 FD
.: 0838 A6 DE D0 92 20 CA FA 00
.: 0840 B0 DD 4C 56 FD 20 81 FA
.: 0848 00 8D 0D 02 A5 FC 8D 0E
.: 0850 02 A9 04 A2 00 00 8D 09
.: 0858 02 8E 0A 02 A9 93 20 D2
.: 0860 FF A9 16 85 B5 20 06 FC
.: 0868 00 20 64 FC 00 85 FB 84
.: 0870 FC C6 B5 D0 F2 A9 91 20
.: 0878 D2 FF 4C 56 FD A0 2C 20
.: 0880 15 FE 20 6A E7 20 CD FD
.: 0888 A2 00 00 A1 FB 20 74 FC
.: 0890 00 48 20 BB FC 00 68 20
.: 0898 D3 FC 00 A2 06 E0 03 D0
.: 08A0 13 AC 1C 02 F0 0E A5 FF
.: 08A8 C9 E8 B1 FB B0 1C 20 5C
.: 08B0 FC 00 88 D0 F2 06 FF 90
.: 08B8 0E BD 51 FF 00 20 45 FD
.: 08C0 00 BD 57 FF 00 F0 03 20
.: 08C8 45 FD 00 CA D0 D4 60 20
.: 08D0 68 FC 00 AA E8 D0 01 C8
.: 08D8 98 20 5C FC 00 8A 86 B4
.: 08E0 20 75 E7 A6 B4 60 AD 1C
.: 08E8 02 38 A4 FC AA 10 01 88
.: 08F0 65 FB 90 01 C8 60 A8 4A
.: 08F8 90 0B 4A B0 17 C9 22 F0
.: 0900 13 29 07 09 80 4A AA BD
.: 0908 00 FF 00 B0 04 4A 4A 4A
.: 0910 4A 29 0F D0 04 A0 80 A9
.: 0918 00 00 AA BD 44 FF 00 85
.: 0920 FF 29 03 8D 1C 02 98 29
.: 0928 8F AA 98 A0 03 E0 8A F0
.: 0930 0B 4A 90 08 4A 4A 09 20
.: 0938 88 D0 FA C8 88 D0 F2 60
.: 0940 B1 FB 20 5C FC 00 A2 01
.: 0948 20 A1 FA 00 CC 1C 02 C8
.: 0950 90 F0 A2 03 CC 09 02 90
.: 0958 F0 60 A8 B9 5E FF 00 8D
.: 0960 0B 02 B9 9E FF 00 8D 0C
.: 0968 02 A9 00 00 A0 05 0E 0C
.: 0970 02 2E 0B 02 2A 88 D0 F6
.: 0978 69 3F 20 D2 FF CA D0 EA
.: 0980 4C CD FD 20 81 FA 00 20
.: 0988 97 E7 20 92 FA 00 20 97
.: 0990 E7 A9 04 A2 00 00 8D 09
.: 0998 02 8E 0A 02 20 D0 FD 20
.: 09A0 0B FC 00 20 64 FC 00 85
.: 09A8 FB 84 FC 20 01 F3 F0 05
.: 09B0 20 CA FA 00 B0 E9 4C 56
.: 09B8 FD 20 81 FA 00 A9 03 85
.: 09C0 B5 20 EB E7 20 A7 FD D0
.: 09C8 F8 AD 0D 02 85 FB AD 0E
.: 09D0 02 85 FC 4C E7 FB 00 CD
.: 09D8 0A 02 F0 03 20 D2 FF 60
.: 09E0 A9 03 A2 24 8D 09 02 8E
.: 09E8 0A 02 20 D0 FD 78 AD FA
.: 09F0 FF 00 85 90 AD FB FF 00
.: 09F8 85 91 A9 A0 8D 4E E8 CE
.: 0A00 13 E8 A9 2E 8D 48 E8 A9
.: 0A08 00 00 8D 49 E8 AE 06 02
.: 0A10 9A 4C F1 FE 20 7B FC 68
.: 0A18 8D 05 02 68 8D 04 02 68
.: 0A20 8D 03 02 68 8D 02 02 68
.: 0A28 8D 01 02 68 8D 00 00 02
.: 0A30 BA 8E 06 02 58 20 D0 FD
.: 0A38 20 BF FD 85 B5 A0 00 00
.: 0A40 20 9A FD 20 CD FD AD 00
.: 0A48 00 02 85 FC AD 01 02 85
.: 0A50 FB 20 6A E7 20 0E FC 00
.: 0A58 20 01 F3 C9 F7 F0 F9 20
```

```
.: 0A60 01 F3 D0 03 4C 56 FD C9
.: 0A68 FF F0 F4 4C 5B FD 00 20
.: 0A70 81 FA 00 20 97 E7 8E 11
.: 0A78 02 A2 03 20 79 FA 00 48
.: 0A80 CA D0 F9 A2 03 68 38 E9
.: 0A88 3F A0 05 4A 6E 11 02 6E
.: 0A90 10 02 88 D0 F6 CA D0 ED
.: 0A98 A2 02 20 CF FF C9 0D F0
.: 0AA0 1E C9 20 F0 F5 20 F7 FE
.: 0AA8 00 B0 0F 20 CB E7 A4 FB
.: 0AB0 84 FC 85 FB A9 30 9D 10
.: 0AB8 02 E8 9D 10 02 E8 D0 DB
.: 0AC0 8E 0B 02 A2 00 00 86 DE
.: 0AC8 F0 04 E6 DE F0 7B A2 00
.: 0AD0 00 86 B5 A5 DE 20 74 FC
.: 0AD8 00 A6 FF 8E 0C 02 AA BC
.: 0AE0 5E FF 00 BD 9E FF 00 20
.: 0AE8 E0 FE 00 D0 E2 A2 06 E0
.: 0AF0 03 D0 1A AC 1C 02 F0 15
.: 0AF8 A5 FF C9 E8 A9 30 B0 21
.: 0B00 20 E6 FE 00 D0 CA 20 E8
.: 0B08 FE 00 D0 C5 88 D0 EB 06
.: 0B10 FF 90 0B BC 57 FF 00 BD
.: 0B18 51 FF 00 20 E0 FE 00 D0
.: 0B20 B3 CA D0 D0 F0 0A 20 DF
.: 0B28 FE 00 D0 A9 20 DF FE 00
.: 0B30 D0 A4 AD 0B 02 C5 B5 D0
.: 0B38 9D 20 97 E7 AC 1C 02 F0
.: 0B40 2F AD 0C 02 C9 9D D0 20
.: 0B48 20 CA FA 00 90 0B 98 D0
.: 0B50 85 AE 1B 02 10 0B 4C 9A
.: 0B58 FA 00 C8 D0 FA AE 1B 02
.: 0B60 10 F5 CA CA 8A AC 1C 02
.: 0B68 D0 03 B9 FC 00 00 91 FB
.: 0B70 88 D0 F8 A5 DE 91 FB 20
.: 0B78 64 FC 00 85 FB 84 FC A0
.: 0B80 41 20 15 FE 20 6A E7 20
.: 0B88 CD FD 4C D8 FD 00 A8 20
.: 0B90 E6 FE 00 D0 11 98 F0 0E
.: 0B98 86 B4 A6 B5 DD 10 02 08
.: 0BA0 E8 86 B5 A6 B4 28 60 C9
.: 0BA8 30 90 03 C9 47 60 38 60
.: 0BB0 40 02 45 03 D0 08 40 09
.: 0BB8 30 22 45 33 D0 08 40 09
.: 0BC0 40 02 45 33 D0 08 40 09
.: 0BC8 40 02 45 B3 D0 08 40 09
.: 0BD0 00 00 22 44 33 D0 8C 44
.: 0BD8 00 00 11 22 44 33 D0 8C
.: 0BE0 44 9A 10 22 44 33 D0 08
.: 0BE8 40 09 10 22 44 33 D0 08
.: 0BF0 40 09 62 13 78 A9 00 00
.: 0BF8 21 81 82 00 00 00 00 59
.: 0C00 4D 91 92 86 4A 85 9D 2C
.: 0C08 29 2C 23 28 24 59 00 00
.: 0C10 58 24 24 00 00 1C 8A 1C
.: 0C18 23 5D 8B 1B A1 9D 8A 1D
.: 0C20 23 9D 8B 1D A1 00 00 29
.: 0C28 19 AE 69 A8 19 23 24 53
.: 0C30 1B 23 24 53 19 A1 00 00
.: 0C38 1A 5B 5B A5 69 24 24 AE
.: 0C40 AE A8 AD 29 00 00 7C 00
.: 0C48 00 15 9C 6D 9C A5 69 29
.: 0C50 53 84 13 34 11 A5 69 23
.: 0C58 A0 D8 62 5A 48 26 62 94
.: 0C60 88 54 44 C8 54 68 44 E8
.: 0C68 94 00 00 B4 08 84 74 B4
.: 0C70 28 6E 74 F4 CC 4A 72 F2
.: 0C78 A4 8A 00 00 AA A2 A2 74
.: 0C80 74 74 72 44 68 B2 32 B2
.: 0C88 00 00 22 00 00 1A 1A 26
.: 0C90 26 72 72 88 C8 C4 CA 26
.: 0C98 48 44 44 A2 C8 54 46 48
.: 0CA0 44 50 2C 41 49 4E 00 00
.: 0CA8 DB FA 00 30 FB 00 5E FB
.: 0CB0 00 D1 FB 00 F8 FC 00 28
.: 0CB8 FD 00 D4 FD 00 4D FD 00
.: 0CC0 55 FD 7F FD 00 4A FA 00
.: 0CC8 33 FA 00 AA AA AA AA AA
.: 0CD0 AA AA AA AA AA AA AA AA
.: 0CD8 AA AA AA AA AA AA AA AA
.: 0CE0 AA AA AA AA AA AA AA AA
.: 0CE8 AA AA AA AA AA AA AA AA
.: 0CF0 AA AA AA AA AA AA AA AA
.: 0CF8 AA AA AA AA AA AA AA AA
.: 0D00 AA AA AA AA AA AA AA AA
.: 0D08 AA AA AA AA AA AA AA AA
.: 0D10 AA AA AA AA AA AA AA AA
.: 0D18 AA AA AA AA AA AA AA AA
```

Type in the following program. It will make the debugging of the long list of machine code much simpler. Every 64 bytes of machine code have been added together to give a number. This is stored in the data statements. When the program is run with the machine code resident it will add groups of 64 bytes together and compare the result with the checksum. It will print out a number which represents the blocks which match the checksum. If a block fails the test then the program will print an error message giving the hexadecimal boundaries for you to check through. Once again this is for BASIC2 PETs.

```
10 I=1536:E=63
20 READD:FORN=ITOI+E:V=V+PEEK(N):NEXT:
   IFV=DTHENV=0:I=I+E:K=K+1:PRINTK:
   GOTO20
30 PRINT"ERROR IN RANGE";:GOSUB120:
   PRINT"TO";:I=I+E:K=K+1:GOSUB120:
   PRINT:GOTO20
120 N=I:K=16:PRINT" $";:D=-LOG(N)/LOG(K
    ):D%=D-(D<>INT(D)):FORR=D%TO0
130 P=K↑(-R):Q%=N/P:PRINTCHR$(Q%+48-7*(
    Q%>9));:N=N-Q%*P:NEXT:PRINT" ";:
    RETURN
1000 DATA5706,9173,8681,8612,8602,8722,
     8317,8846,8705,7976,8074
1010 DATA7840,6479,6789,7491,8303,6812,
     8060,7689,8698,9147,7643
1020 DATA8941,4421,4208,5488,6410,5704,
     8571
```

When the checksums give no errors the machine code can be linked to the BASIC program in the following manner. Load the machine code for Supermon and then type NEW. Now load the BASIC program. Type POKE42,203:POKE43,12. This will link the BASIC program to the machine code. Save Supermon like any other BASIC program to either disk or tape.

The following Supermon is for use on BASIC1 PETs, it was converted by David Hills and includes TIM which is not available in ROM for the 8k PET. This is a slightly older version of Supermon than the one given for the new ROM PETs. Care must be taken when using the simple assembler as it has a tendency to crash if too much code is input at any one time due to inefficient use of the stack. The printing disassembler has not been implemented.

```
.: 16F6 A9 06 8D 1B 02 A9 17 8D
.: 16FE 1C 02 00 00 00 00 00 00
.: 1706 A9 42 85 21 D8 4A 68 85
.: 170E 1E 68 85 1D 68 85 1C 68
.: 1716 85 1B 68 69 FF 85 19 68
.: 171E 69 FF 85 1A BA 86 1F 58
.: 1726 20 BC 17 A6 21 A9 2A 20
.: 172E EC 18 A9 52 85 0D D0 1B
.: 1736 A9 00 85 CA 85 0D 85 0A
.: 173E 20 BC 17 A9 2E 20 D2 FF
.: 1746 20 5A 19 C9 2E F0 F9 C9
.: 174E 20 F0 F5 A2 07 DD CC 17
.: 1756 D0 0F A5 20 85 0E 86 20
.: 175E BD D4 17 48 BD DC 17 48
.: 1766 60 CA 10 E9 4C 5D 1A 38
.: 176E A5 13 E5 11 85 0B A5 14
.: 1776 E5 12 A8 05 0B 60 A5 11
.: 177E 85 19 A5 12 85 1A 60 85
.: 1786 21 A0 00 20 04 19 B1 11
.: 178E 20 DD 18 20 C1 17 C6 21
.: 1796 D0 F1 60 20 28 19 90 0D
.: 179E A2 00 81 11 C1 11 F0 05
.: 17A6 68 68 4C 65 17 20 C1 17
.: 17AE C6 21 60 A9 1B 85 11 A9
.: 17B6 00 85 12 A9 05 60 A9 0D
.: 17BE 4C D2 FF E6 11 D0 06 E6
.: 17C6 12 D0 02 E6 0A 60 3A 3B
.: 17CE 52 4D 47 58 4C 53 18 18
.: 17D6 17 18 18 18 19 19 8B 7B
.: 17DE F6 28 A1 C7 68 68 20 50
.: 17E6 43 20 20 53 52 20 41 43
.: 17EE 20 58 52 20 59 52 20 53
.: 17F6 50 A5 0D D0 06 20 BC 17
.: 17FE 20 01 19 20 01 19 A2 00
.: 1806 BD E4 17 20 D2 FF E8 E0
.: 180E 13 D0 F5 20 BC 17 A2 2E
.: 1816 A9 3B 20 EC 18 20 01 19
.: 181E 20 D2 18 20 B1 17 20 85
.: 1826 17 F0 4D 20 5A 19 20 19
.: 182E 19 90 48 20 09 19 20 5A
.: 1836 19 20 19 19 90 3D 20 09
.: 183E 19 A0 00 B9 14 1A 30 06
.: 1846 20 D2 FF C8 D0 F5 29 7F
.: 184E 20 D2 FF 20 2A F3 F0 20
.: 1856 A6 0A D0 1C 20 6D 17 90
.: 185E 17 20 BC 17 A2 2E A9 3A
.: 1866 20 EC 18 20 01 19 20 CE
.: 186E 18 A9 08 20 85 17 F0 DB
.: 1876 4C 36 17 4C 65 17 20 28
.: 187E 19 20 19 19 90 03 20 7C
.: 1886 17 20 B1 17 D0 0A 20 28
.: 188E 19 20 19 19 90 E5 A9 08
.: 1896 85 21 20 5A 19 20 99 17
.: 189E D0 F8 F0 D4 20 CF FF C9
.: 18A6 0D F0 0C C9 20 D0 CC 20
.: 18AE 19 19 90 03 20 7C 17 A6
.: 18B6 1F 9A A5 1A 48 A5 19 48
.: 18BE A5 1B 48 A5 1C A6 1D A4
.: 18C6 1E 40 A6 1F 9A 4C 8B C3
.: 18CE A2 01 D0 02 A2 09 B5 10
.: 18D6 48 B5 11 20 DD 18 68 48
.: 18DE 4A 4A 4A 4A 20 F5 18 AA
.: 18E6 68 29 0F 20 F5 18 48 8A
.: 18EE 20 D2 FF 68 4C D2 FF 18
.: 18F6 69 06 69 F0 90 02 69 06
.: 18FE 69 3A 60 20 04 19 A9 20
.: 1906 4C D2 FF A2 02 B5 10 48
.: 190E B5 12 95 10 68 95 12 CA
.: 1916 D0 F3 60 20 28 19 90 02
.: 191E 85 12 20 28 19 90 02 85
.: 1926 11 60 A9 00 85 0F 20 5A
.: 192E 19 C9 20 D0 09 20 5A 19
.: 1936 C9 20 D0 0E 18 60 20 4F
.: 193E 19 0A 0A 0A 0A 85 0F 20
.: 1946 5A 19 20 4F 19 05 0F 38
.: 194E 60 C9 3A 08 29 0F 28 90
```

```
.: 1956 02 69 08 60 20 CF FF C9
.: 195E 0D D0 F8 68 68 4C 36 17
.: 1966 4C 65 17 20 5A 19 A9 00
.: 196E 85 EE 85 FA A9 23 85 F9
.: 1976 20 28 19 29 0F 85 F1 20
.: 197E 5A 19 A2 00 20 CF FF C9
.: 1986 2C F0 55 C9 0D F0 0B E0
.: 198E 10 F0 F1 95 23 E6 EE E8
.: 1996 D0 EA A5 20 C9 06 D0 C8
.: 199E A2 00 8E 0B 02 A5 F1 D0
.: 19A6 03 4C 65 17 C9 03 B0 F9
.: 19AE 20 67 F6 20 3B F8 20 FF
.: 19B6 F3 A5 EE F0 08 20 95 F4
.: 19BE D0 08 4C 65 17 20 AE F5
.: 19C6 F0 F8 20 4D F6 20 22 F4
.: 19CE 20 8A F8 20 13 F9 AD 0C
.: 19D6 02 29 10 D0 E5 4C 36 17
.: 19DE 20 19 19 A5 11 85 F7 A5
.: 19E6 12 85 F8 20 CF FF C9 20
.: 19EE F0 F9 C9 0D F0 A4 C9 2C
.: 19F6 F0 03 4C 66 19 20 19 19
.: 19FE A5 11 85 E5 A5 12 85 E6
.: 1A06 A5 20 C9 06 F0 92 A2 00
.: 1A0E 20 B1 F6 4C 36 17 0D 20
.: 1A16 20 20 20 20 20 20 20 20
.: 1A1E 20 30 20 20 31 20 20 32
.: 1A26 20 20 33 20 20 34 20 20
.: 1A2E 35 20 20 36 20 20 B7 00
.: 1A36 00 00 98 48 20 BC 17 68
.: 1A3E A2 2E 20 EC 18 4C 01 19
.: 1A46 AD FE 1F 85 86 AD FF 1F
.: 1A4E 85 87 4C F6 16 00 00 A9
.: 1A56 3F 20 D2 FF 4C 36 17 A2
.: 1A5E 08 DD DE 1F D0 0E 86 20
.: 1A66 8A 0A AA BD E9 1F 48 BD
.: 1A6E E8 1F 48 60 CA 10 EA 4C
.: 1A76 55 1A A2 02 2C A2 00 B4
.: 1A7E 11 D0 08 B4 12 D0 02 E6
.: 1A86 0A D6 12 D6 11 60 20 5A
.: 1A8E 19 C9 20 F0 F9 60 A9 00
.: 1A96 8D 0F 00 20 8C 1A 20 2F
.: 1A9E 19 20 1C 19 90 09 60 20
.: 1AA6 5A 19 20 19 19 B0 DE 4C
.: 1AAE 55 1A 20 04 19 CA D0 FA
.: 1AB6 60 E6 13 D0 02 E6 14 60
.: 1ABE A2 02 B5 10 48 BD 2B 00
.: 1AC6 95 10 68 9D 2B 00 CA D0
.: 1ACE F1 60 AD 2C 00 AC 2D 00
.: 1AD6 4C DD 1A A5 13 A4 14 38
.: 1ADE E5 11 85 EC 98 E5 12 A8
.: 1AE6 05 EC 60 20 94 1A 20 09
.: 1AEE 19 20 A5 1A 20 BE 1A 20
.: 1AF6 A5 1A 20 D9 1A 20 09 19
.: 1AFE 90 15 A6 0A D0 64 20 D0
.: 1B06 1A 90 5F A1 11 81 13 20
.: 1B0E B7 1A 20 C1 17 D0 EB 20
.: 1B16 D0 1A 18 A5 EC 65 13 85
.: 1B1E 13 98 65 14 85 14 20 BE
.: 1B26 1A A6 0A D0 3D A1 11 81
.: 1B2E 13 20 D0 1A B0 34 20 78
.: 1B36 1A 20 7B 1A 4C 27 1B 20
.: 1B3E 94 1A 20 09 19 20 A5 1A
.: 1B46 20 09 19 20 5A 19 20 28
.: 1B4E 19 90 14 85 21 A6 0A D0
.: 1B56 11 20 D9 1A 90 0C A5 21
.: 1B5E 81 11 20 C1 17 D0 EE 4C
.: 1B66 55 1A 4C 36 17 20 94 1A
.: 1B6E 20 09 19 20 A5 1A 20 09
.: 1B76 19 20 5A 19 A2 00 20 5A
.: 1B7E 19 C9 27 D0 14 20 5A 19
.: 1B86 9D 31 00 E8 20 CF FF C9
.: 1B8E 0D F0 22 E0 20 D0 F1 F0
.: 1B96 1C 8E 0F 00 20 2F 19 90
.: 1B9E C6 9D 31 00 E8 20 CF FF
.: 1BA6 C9 0D F0 09 20 28 19 90
.: 1BAE B6 E0 20 D0 EC 86 20 20
```

```
.:  1BB6 BC 17 A2 00 A0 00 B1 11
.:  1BBE DD 31 00 D0 0C C8 E8 E4
.:  1BC6 20 D0 F3 20 CE 18 20 04
.:  1BCE 19 20 C1 17 A6 0A D0 92
.:  1BD6 20 D9 1A B0 DD 4C 36 17
.:  1BDE 20 94 1A 8D 2E 00 A5 12
.:  1BE6 8D 2F 00 A9 04 A2 00 85
.:  1BEE 25 86 26 A9 93 20 D2 FF
.:  1BF6 A9 16 85 21 20 10 1C 20
.:  1BFE 6D 1C 85 11 84 12 C6 21
.:  1C06 D0 F2 A9 91 20 D2 FF 4C
.:  1C0E 36 17 A0 2C 20 38 1A 20
.:  1C16 CE 18 20 04 19 A2 00 A1
.:  1C1E 11 20 7C 1C 48 20 C2 1C
.:  1C26 68 20 D8 1C A2 06 E0 03
.:  1C2E D0 12 A4 23 F0 0E A5 FF
.:  1C36 C9 E8 B1 11 B0 1C 20 65
.:  1C3E 1C 88 D0 F2 06 FF 90 0E
.:  1C46 BD 4A 1F 20 4D 1D BD 50
.:  1C4E 1F F0 03 20 4D 1D CA D0
.:  1C56 D5 60 20 70 1C AA E8 D0
.:  1C5E 01 C8 98 20 65 1C 8A 86
.:  1C66 20 20 DD 18 A6 20 60 A5
.:  1C6E 23 38 A4 12 AA 10 01 88
.:  1C76 65 11 90 01 C8 60 A8 4A
.:  1C7E 90 0B 4A B0 17 C9 22 F0
.:  1C86 13 29 07 09 80 4A AA BD
.:  1C8E F9 1E B0 04 4A 4A 4A 4A
.:  1C96 29 0F D0 04 A0 80 A9 00
.:  1C9E AA BD 3D 1F 85 FF 29 03
.:  1CA6 85 23 98 29 8F AA 98 A0
.:  1CAE 03 E0 8A F0 0B 4A 90 08
.:  1CB6 4A 4A 09 20 88 D0 FA C8
.:  1CBE 88 D0 F2 60 B1 11 20 65
.:  1CC6 1C A2 01 20 B0 1A C4 23
.:  1CCE C8 90 F1 A2 03 C4 25 90
.:  1CD6 F2 60 A8 B9 57 1F 8D 2C
.:  1CDE 00 B9 97 1F 8D 2D 00 A9
.:  1CE6 00 A0 05 0E 2D 00 2E 2C
.:  1CEE 00 2A 88 D0 F6 69 3F 20
.:  1CF6 D2 FF CA D0 EA 4C 04 19
.:  1CFE 20 94 1A 20 C1 17 20 C1
.:  1D06 17 20 09 19 20 A5 1A 20
.:  1D0E 09 19 20 01 19 20 D9 1A
.:  1D16 90 09 98 D0 13 A5 EC 30
.:  1D1E 0F 10 07 C8 D0 0A A5 EC
.:  1D26 10 06 20 DD 18 4C 36 17
.:  1D2E 4C 55 1A 20 94 1A A9 03
.:  1D36 85 21 20 5A 19 20 99 17
.:  1D3E D0 F8 AD 2E 00 85 11 AD
.:  1D46 2F 00 85 12 4C F1 1B C5
.:  1D4E 26 F0 03 20 D2 FF 60 A9
.:  1D56 03 A2 24 85 25 86 26 20
.:  1D5E BC 17 78 A9 84 8D 19 02
.:  1D66 A9 1D 8D 1A 02 A9 A0 8D
.:  1D6E 4E E8 CE 13 E8 A9 28 8D
.:  1D76 48 E8 A9 00 8D 49 E8 AE
.:  1D7E 1F 00 9A 4C B8 18 20 FB
.:  1D86 FC 68 8D 1E 00 68 8D 1D
.:  1D8E 00 68 8D 1C 00 68 8D 1B
.:  1D96 00 68 8D 19 00 68 8D 1A
.:  1D9E 00 BA 8E 1F 00 58 20 BC
.:  1DA6 17 20 B1 17 85 21 A0 00
.:  1DAE 20 8C 17 20 04 19 AD 1A
.:  1DB6 00 85 12 AD 19 00 85 11
.:  1DBE 20 CE 18 20 18 1C 20 2A
.:  1DC6 F3 C9 F7 F0 F9 20 2A F3
.:  1DCE D0 03 4C 36 17 C9 FF F0
.:  1DD6 F4 4C 60 1D 00 20 94 1A
.:  1DDE 20 09 19 8E 32 00 A2 03
.:  1DE6 20 8C 1A 48 CA D0 F9 A2
.:  1DEE 03 68 38 E9 3F A0 05 4A
.:  1DF6 6E 32 00 6E 31 00 88 D0
.:  1DFE F6 CA D0 ED A2 02 20 CF
.:  1E06 FF C9 0D F0 1E C9 20 F0
.:  1E0E F5 20 F0 1E B0 0F 20 3C
.:  1E16 19 A4 11 84 12 85 11 A9
.:  1E1E 30 9D 31 00 E8 9D 31 00
.:  1E26 E8 D0 DB 8E 2C 00 A2 00
.:  1E2E 86 0A A2 00 86 21 A5 0A
.:  1E36 20 7C 1C A6 FF 8E 2D 00
.:  1E3E AA BD 97 1F 20 D5 1E BD
.:  1E46 57 1F 20 D5 1E A2 06 E0
.:  1E4E 03 D0 12 A4 23 F0 0E A5
.:  1E56 FF C9 E8 A9 30 B0 1D 20
.:  1E5E D2 1E 88 D0 F2 06 FF 90
.:  1E66 0E BD 4A 1F 20 D5 1E BD
.:  1E6E 50 1F F0 03 20 D5 1E CA
.:  1E76 D0 D5 F0 06 20 D2 1E 20
.:  1E7E D2 1E AD 2C 00 C5 21 D0
.:  1E86 59 20 09 19 A4 23 F0 2B
.:  1E8E AD 2D 00 C9 9D D0 1C 20
.:  1E96 D9 1A 90 09 98 D0 4A A6
.:  1E9E EC 30 46 10 07 C8 D0 41
.:  1EA6 A6 EC 10 3D CA CA 8A A4
.:  1EAE 23 D0 03 B9 12 00 91 11
.:  1EB6 88 D0 F8 A5 0A 91 11 20
.:  1EBE 6D 1C 85 11 84 12 A0 41
.:  1EC6 20 38 1A 20 CE 18 20 04
.:  1ECE 19 4C DE 1D 20 D5 1E 86
.:  1ED6 20 A6 21 DD 31 00 F0 0C
.:  1EDE 68 68 E6 0A F0 03 4C 30
.:  1EE6 1E 4C 55 1A E8 86 21 A6
.:  1EEE 20 60 C9 30 90 03 C9 47
.:  1EF6 60 38 60 40 02 45 03 D0
.:  1EFE 08 40 09 30 22 45 33 D0
.:  1F06 08 40 09 40 02 45 33 D0
.:  1F0E 08 40 09 40 02 45 B3 D0
.:  1F16 08 40 09 00 22 44 33 D0
.:  1F1E 8C 44 00 11 22 44 33 D0
.:  1F26 8C 44 9A 10 22 44 33 D0
.:  1F2E 08 40 09 10 22 44 33 D0
.:  1F36 08 40 09 62 13 78 A9 00
.:  1F3E 21 81 82 00 00 59 4D 91
.:  1F46 92 86 4A 85 9D 2C 29 2C
.:  1F4E 23 28 24 59 00 58 24 24
.:  1F56 00 1C 8A 1C 23 5D 8B 1B
.:  1F5E A1 9D 8A 1D 23 9D 8B 1D
.:  1F66 A1 00 29 19 AE 69 A8 19
.:  1F6E 23 24 53 1B 23 24 53 19
.:  1F76 A1 00 1A 5B 5B A5 69 24
.:  1F7E 24 AE AE A8 AD 29 00 7C
.:  1F86 00 15 9C 6D 9C A5 69 29
.:  1F8E 53 84 13 34 11 A5 69 23
.:  1F96 A0 D8 62 5A 48 26 62 94
.:  1F9E 88 54 44 C8 54 68 44 E8
.:  1FA6 94 00 B4 08 84 74 B4 28
.:  1FAE 6E 74 F4 CC 4A 72 F2 A4
.:  1FB6 8A 00 AA A2 A2 74 74 74
.:  1FBE 72 44 68 B2 32 B2 00 22
.:  1FC6 00 1A 1A 26 26 72 72 88
.:  1FCE C8 C4 CA 26 48 44 44 A2
.:  1FD6 C8 04 22 10 20 2D 2F 33
.:  1FDE 54 46 48 44 43 2C 41 49
.:  1FE6 4E 00 E8 1A 3C 1B 6A 1B
.:  1FEE DD 1B FD 1C 30 1D DA 1D
.:  1FF6 54 1D 55 FD 84 1D 5D 1A
.:  1FFE F6 16 AA AA AA AA AA AA
```

The following program is the checksum test for the BASIC1 Supermon. POKE135,20:POKE134,0 will stop the checksum program intruding into the machine code.

```
10 I=5878:E=63
20 READC:FORN=ITOI+E:V=V+PEEK(N):NEXT:
   IFV=CTHENV=0:I=I+E:K=K+1:PRINTK:
   GOTO20
30 PRINT"ERROR IN RANGE";:GOSUB120:
   PRINT"TO";:I=I+E:K=K+1:GOSUB120:
   PRINT:GOTO20
```

```
120 N=I:K=16:PRINT" $";:D=-LOG(N)/LOG(K
    ):D%=D-(D<>INT(D)):FORR=D%TO0
130 P=K↑(-R):Q%=N/P:PRINTCHR$(Q%+48-7*(
    Q%>9));:N=N-Q%*P:NEXT:PRINT" ";:
    RETURN
1000 DATA5722,7263,5701,5516,5653,5625,
     5808,5832,6080,4825,8071,7773,5980

1010 DATA5799,6017,6563,5667,4987,5676,
     6957,5619,7192,6129,7310,5453,5766

1020 DATA6389,5895,6757,6933,7110,6253,
     5309,4620,5250,7179,4649,3702,3976

1030 DATA3607,3677,3772,3595,3676,3502,
     3329,3246,3310,3312,3205,3266,3276

1040 DATA3271,3267,3259,3255,3252,3271,
     3298,3290,3279,3300,3312,3317,3602

1050 DATA2057,1848,2386,2935,2154,2048,
     2048,2048,2048,2048,2048,2048,2048
```

The following program was written to produce the checksums in the above programs. It is a very simple 'self-writing' program to produce DATA statements starting from line 1000. The program is fairly crude but it is simple to modify it to produce data statements of any area of RAM. It stops when a checksum of 10880 is found. This checksum will only be produced when there are 64 consecutive bytes of the code $AA or 170. $AA is the byte which is read into RAM when the PET runs through its power up procedure.

```
10 REM CHECK-SUM PROGRAM
15 REM BY DAVE MIDDLETON
20 REM STARTS AT $7000. STOPS WHEN 170
   FILLS ONE BLOCK OF 64 BYTES
100 EN=0: SR=28672: LN=1000
105 A$=STR$(LN)+" DATA"
110 CS=0: FOR CO=SR TO SR+63: CS=CS+
    PEEK(CO): NEXT: IF CS=10880 THEN EN
    =1
120 A$=A$+STR$(CS):IFLEN(A$)<70THENA$=A
    $+",":GOTO150
130 PRINT"■■■■■";A$:PRINT"LN="LN+10":SR
    ="SR+63":EN="EN":IF EN=0 GOTO105■"
135 POKE158,2:POKE623,13:POKE624,13:
    STOP
150 SR=SR+63: GOTO110
```

IEEE

IEEE BUS HANDSHAKE ROUTINE IN MACHINE LANGUAGE

To use the IEEE-488 bus on the PET at maximum speed it is necessary to use machine language rather than BASIC 'INPUT' and PRINT. The routine given here has been used with an HP3437A systems voltmeter to reach data transfer speeds of over 5000 bytes per second, corresponding to 2500 voltage readings in 2-byte packed binary format or 625 readings in 8-byte ASCII format. The best speed attained in BASIC is 75 readings per second transferred as character strings.

THE IEEE BUS

Details of the IEEE-488 bus are given in the PET Users Handbook, but some clarification of the register addresses on page 120 of the handbook is helpful. These are:

HEX	DECIMAL	BITS	IEEE	DIRECTION
E820	59424	0-7	DIO 1-8	from bus
E822	59426	0-7	DIO 1-8	to bus;PET controlled
E821	59425	3	NDAC	PETcontrolled
E823	59427	3	DAV	PETcontrolled
E840	59456	0	NDAC	from bus
		1	NRFD	PETcontrolled
		2	ATN	PETcontrol ed
		6	NRFD	from bus
		7	DAV	from bus

Note that on the IEEE bus, 'high' is logic false and 'low' is logic true; and that the data bus must be left with all bits 'high' when PET has finished to avid confusion of data put on to the bus by other devices.

THE PROGRAM

The program controls a given number of data transfers, each of 8 bytes, from the HP3437A to the PET. Each one is preceded by a trigger (GET - group execute trigger) on the IEEE bus, and the HP3437A must be correctly addressed as a 'talker' or a 'listener' at all times by sending MTA (my talk address) or MLA (my listen address) before transfers as appropriate. The sending of messages (GET, MTA, MLA, etc) or data is controlled by the ATN line; ATN is true when messages are being sent.

The program and returned data are held in the top 2K of memory; this is hidden from BASIC using POKE 134,255:POKE 135,?23 as the first line of the BASIC control program. The number of readings required is POKEd into 6400 , then control passed to the machine language program by SYS(6144). The data bytes coming in on the IEEE bus are stored in locations 6401 onwards; these are PEEKed out on return to BASIC, and converted into numbers using the function VAL. As the index register is used for counting, only 256 bytes can be transferred using this program, but it would be easy to modify the program to perform more transfers.

Disassembled listings with comments and a separate listing (for ease of copying into BASIC DATA statements!) are given.

```
LINE# LOC   CODE        LINE

0001  0000                          ;*****************************
0002  0000                          ;*
0003  0000                          ;* IEEE - HAND SHAKE ROUTINE
0004  0000                          ;*
0005  0000                          ;*****************************
0006  0000                          ;
0007  0000                          ;VARIABLES
0008  0000                          ;
0009  0000             INOUTB =$E840      ;PIA B
0010  0000             LOC1   =$01      ;USER LOCATIONS
0011  0000             LOC2   =$02      ;
0012  0000             PIA2   =$E821    ;IEEE PORT
0013  0000             PIA    =$E823    ;
0014  0000             COUNT  =$1900      ;COUNTER 255 MAX
0015  0000             MEM    =$1900    ;RAM STORAGE
0016  0000             DATAF  =$E820      ;DATA FROM IEEE
0017  0000             DATAT  =$E822      ;DATA TO IEEE
0018  0000                                ;
```

```
0019  0000                                    ;START OF MAIN PROGRAM
0020  0000                        *=$1800
0021  1800                                    ;
0022  1800  A2 00                 LDX #$00    ;PREPARE INDEX REGISTER
0023  1802  A9 FB        ATN      LDA #$FB     ;SET ATN LOW
0024  1804  2D 40 E8              AND INOUTB
0025  1807  4D 40 E8              EOR INOUTB
0026  180A  A9 28                 LDA #$28    ;MLA (28 FOR THIS DEVICE)
0027  180C  85 01                 STA LOC1
0028  180E  20 68 18              JSR HANBUS  ;HANDSHAKE INTO BUS
0029  1811  A9 08                 LDA #$08    ;GET
0030  1813  85 01                 STA LOC1
0031  1815  20 68 18              JSR HANBUS  ;HANDSHAKE
0032  1818  A9 FD                 LDA #$FD    ;SET NRFD LOW - RECIEVE
0033  181A  2D 40 E8              AND INOUTB
0034  181D  8D 40 E8              STA INOUTB
0035  1820  A9 F7                 LDA #$F7    ;AND NDAC LOW AS WELL
0036  1822  2D 21 E8              AND PIA2
0037  1825  8D 21 E8              STA PIA2
0038  1828  A9 04                 LDA #$04    ;SET ATN HIGH
0039  182A  0D 40 E8              ORA INOUTB
0040  182D  8D 40 E8              STA INOUTB
0041  1830  A0 08                 LDY #$08    ;READY TO COUNT 8 BYTES
0042  1832  20 93 18     LOOP     JSR BUSHAN   ;HANDSHAKE DATA FROM BUS
0043  1835  A5 02                 LDA LOC2    ;RESULT TO A
0044  1837  9D 00 19              STA MEM,X   ;STORE IN MEM+X
0045  183A  E8                    INX
0046  183B  88                    DEY
0047  183C  D0 F4                 BNE LOOP    ;JUMP IF NOT ZERO
0048  183E  A9 FB                 LDA #$FB    ;SET ATN LOW
0049  1840  2D 40 E8              AND INOUTB
0050  1843  8D 40 E8              STA INOUTB
0051  1846  A9 02                 LDA #$02    ;SET NRFD HIGH
0052  1848  0D 40 E8              ORA INOUTB
0053  184B  8D 40 E8              STA INOUTB
0054  184E  A9 08                 LDA #$08    ;SET NDAC HIGH
0055  1850  0D 21 E8              ORA PIA2
```

```
IEEE*......PAGE 0002

LINE# LOC   CODE        LINE
```

```
0056  1853  8D 21 E8              STA PIA2
0057  1856  A9 5F                 LDA #$5F    ;UNT
0058  1858  85 01                 STA LOC1
0059  185A  20 68 18              JSR HANBUS  ;HANDSHAKE TO BUS
0060  185D  A9 04                 LDA #$04    ;SET ATN HIGH
0061  185F  0D 40 E8              ORA INOUTB
0062  1862  CE 00 19              DEC COUNT   ;DECREMENT COUNT
0063  1865  D0 9B                 BNE ATN     ;JUMP IF NOT ZERO
0064  1867  60                    RTS         ;RETURN TO BASIC PROGRAM
0065  1868                                    ;
0066  1868                                   ;SUBROUTINE TO HANDSHAKE  DATA
0067  1868                                   ;INTO THE IEEE BUS
0068  1868                                    ;
0069  1868  AD 40 E8     HANBUS LDA INOUTB    ;NRFD?
0070  186B  29 40                 AND #$40
0071  186D  F0 F9                 BEQ HANBUS ;JUMP BACK IF NOT READY
0072  186F  A5 01                 LDA LOC1    ;GET DATA BYTE
0073  1871  49 FF                 EOR #$FF    ;COMPLEMENT IT
0074  1873  8D 22 E8              STA DATAT   ;SEND IT TO BUS
0075  1876  A9 F7                 LDA #$F7    ;SET DAV LOW
0076  1878  2D 23 E8              AND PIA
0077  187B  8D 23 E8              STA PIA
0078  187E  AD 40 E8     NDAC     LDA INOUTB   ;NDAC?
0079  1881  29 01                 AND #$01
0080  1883  F0 F9                 BEQ NDAC    ;JUMP BACK IF NOT ACCEPTED
0081  1885  A9 08                 LDA #$08    ;SET DAV HIGH
0082  1887  0D 23 E8              ORA PIA
0083  188A  8D 23 E8              STA PIA
```

```
0084  188D  A9 FF          LDA #$FF       ;255 INTO BUS
0085  188F  8D 22 E8       STA DATAT
0086  1892  60             RTS            ;RETURN TO MAIN
0087  1893                                ;
0088  1893                               ;SUBROUTINE TO HANDLE DATA
0089  1893                               ;HANDSHAKE FRO BUS
0090  1893                                ;
0091  1893  A9 02       BUSHAN LDA #$02     ;SET NRFD HIGH
0092  1895  0D 40 E8          ORA INOUTB
0093  1898  8D 40 E8          STA INOUTB
0094  189B  AD 40 E8    DAV   LDA INOUTB    ;DAV?
0095  189E  29 80             AND #$80
0096  18A0  D0 F9             BNE DAV        ;JUMP BACK IF NOT VALID
0097  18A2  AD 20 E8          LDA DATAF      ;GET DATA BYTE
0098  18A5  49 FF             EOR #$FF       ;COMPLEMENT
0099  18A7  85 02             STA LOC2       ;STORE
0100  18A9  A9 FD             LDA #$FD       ;SET NRFD LOW
0101  18AB  2D 40 E8          AND INOUTB
0102  18AE  8D 40 E8          STA INOUTB
0103  18B1  A9 08             LDA #$08       ;SET NDAC HIGH
0104  18B3  0D 21 E8          ORA PIA2
0105  18B6  8D 21 E8          STA PIA2
0106  18B9  AD 40 E8    DAVH  LDA INOUTB    ;DAV HIGH?
0107  18BC  29 80             AND #$80
0108  18BE  F0 F9             BEQ DAVH       ;JUMP BACK IF NOT
0109  18C0  A9 F7             LDA #$F7       ;SET NDAC LOW
0110  18C2  2D 21 E8          AND PIA2
```



```
LINE# LOC.  CODE        LINE

0111  18C5  8D 21 E8          STA PIA2
0112  18C8  A9 FF             LDA #$FF       ;255 INTO BUS
0113  18CA  8D 22 E8          STA DATAT
0114  18CD  60                RTS            ;RETURN TO MAIN
```

The program was written originally on the authors own machine code handler and assembler but for the sake of conformity this has been converted for use on the Commodore Assembler System.

John A. Cooke
Department of Astronomy
University of Edinburgh
Royal Observatory
Edinburgh EH9 3HJ

Articles reprinted by kind permission of
KILOBAUD MICROCOMPUTING MAGAZINE

Get Your PET on the IEEE 488 Bus

by Gregory Yob

Get Your Pet on The IEEE 488 Bus

This 3-part odyssey takes you along route 488. The first stop is here . . . tickets, please.

Gregory Yob
Box 354
Palo Alto, CA 94301

Perhaps the most obscure Commodore PET feature is its IEEE 488 (or HPIB or GPIB) interface. This three-part article describes the rudiments of the 488 bus and how to use your PET to communicate with instruments having the 488 interface. Several working examples with Hewlett-Packard equipment are shown. (HP lent me several 488-compatible instruments to prepare this article.)

If you just want your PET to talk to that costly instrument on your bench, skip this month's installment and start next time with part 2. The first two parts of this article will sketch the prerequisites and give you enough information to track down bugs on your own.

What's a 488 Bus?

In 1972, engineers – some with Hewlett-Packard – proposed a method of joining many instruments in a standardized way to help automate lab and test measurements. This resulted in the IEEE Standard 488-1975, which describes how to connect as many as 15 instruments on the same cable.

HP and several other laboratory-instrument manufacturers then offered the IEEE 488 scheme as an option. Presently, several hundred instruments have the 488 capability; Commodore used to provide a 5-page list of these. The PET was later designed with the instrumentation and control market in mind, so the IEEE 488 interface was put into the PET.

Before the introduction of the PET, instruments capable of controlling the 488 bus cost several thousand dollars. Now the PET often costs less than the instruments it controls. Some 488 manufacturers have trouble adjusting to this – their customers balk at the idea of purchasing an $800 microcomputer to control a $30,000 instrument!

Now one connector joins the PET to many peripherals. You don't need a separate interface and connector for each new gadget. Commodore's printer and disk are designed to use the PET's 488 interface.

Physical Aspects

A PET and a 488-compatible device have different connectors. Your first project is to wire a cable to tie the two machines together.

Fig. 1 shows the location of the IEEE 488 connector on the back of the PET, and Fig. 2 describes the pins and connectors used for the PET and the IEEE 488. I used a 20-conductor ribbon cable and tied the

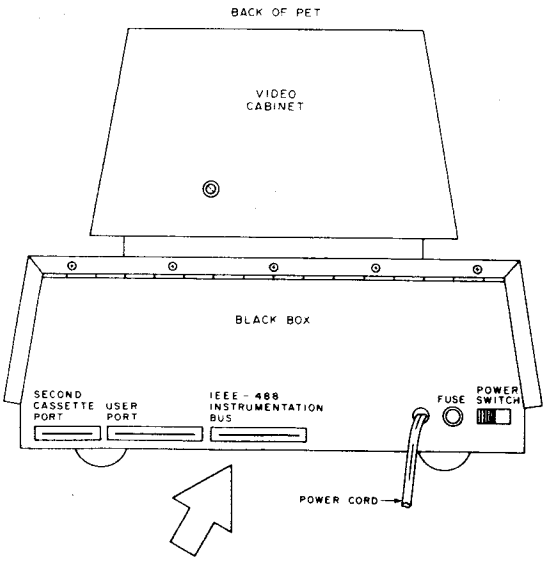


Fig. 1. Location of PET IEEE 488 port on the back of the PET next to the power switch and fuse.

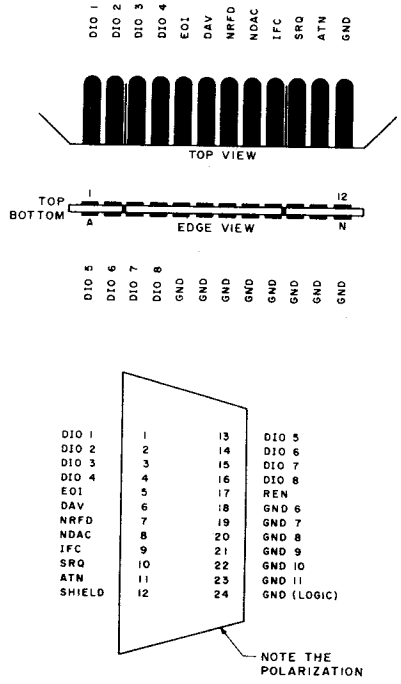


Fig. 2. Pin-outs and connectors for the IEEE 488.

grounds together into the four lines left over after I connected the signal wires.

When making the cable, bear in mind that there are strict limits to cable lengths:

1. The maximum distance between two devices is 5 meters.
2. The longest distance from one end of your setup to the other is 20 meters.
3. A maximum of 15 devices, including the PET, can be hooked together.

It is also wise to avoid electrically noisy areas; don't drape your IEEE 488 cable over your TV set.

If more than one device is connected to the 488, you must use extension cables. HP has cables for about $50. If you want to make your own, consult the two configurations in Fig. 3. The 488 instruments always have a female connector, so have an excess of male connectors on your cables.

Electrically, the 488 bus works on an active-low principle. Fig. 4 shows a circuit similar to a 488 bus line. When all the switches are open, the voltmeter will show 5 volts, which is the false state (or 0) for the line. If any of the switches are closed, the line is grounded, and the voltmeter shows zero volts, or the true state.

This peculiar arrangement permits several devices to be connected to the same line. If any one of them has a switch closed, the line is true. Devices frequently operate at different speeds, and when each device is ready, it opens its switch. However, the line remains true (low) until the slowest device opens its switch.

IEEE Blinkin Lites Display

It is always convenient to have a display and switches to perform a front panel function when you debug interfaces. I built a box, which I call the 488 Blinkin Lites, to display the states of each of the IEEE 488 lines and some switches to force lines low if needed. Fig. 5 shows the circuit, and Fig. 6 is a sketch of my box.

Each line is pulled up to +5 volts with a 10k resistor—the high value was chosen to minimize the load on the 488 bus. The switches can override any line when they are closed to ground. Though the PET doesn't use all the IEEE 488 lines, future machines will—so I put them all in my box.

If you build this box, don't use the PET's +5 volts from the tape port—the LEDs draw 170 mA, which is too much for the PET. Provide a connector to the PET's IEEE port and a male and female IEEE connector. This lets you interpose the IEEE Blinkin Lites between the PET and an instrument.

I mounted a 5×7 inch perfboard with 0.10 inch holes into a standard breadboard box and placed a label near each switch/LED combination to identify the IEEE lines. The three ICs are the 7404s used to drive the LEDs. The cable leads to a homemade junction with a PET connector and IEEE male and female connectors. A mini phono jack connects to a separate +5 volt supply (see Fig. 6).

When you plug in the IEEE Blinkin Lites, the LEDs will show the state of the lines—an LED that is off indicates a low line, which is true; an on LED indicates high, which is false.

The IEEE 488 Lines

The IEEE 488 is composed of 16 lines. Eight are for transfer of data, five are for bus management and three are for handshaking. The eight data lines are labeled DIO1 through DIO8, with the most significant bit (MSB) on DIO8. The 488 bus can transfer one byte at a time and is sometimes called byte-parallel.

The five bus-management lines in various combinations and sequences provide many bus facilities, most of which are rarely used:

EOI—End of Message. When a group of bytes is sent via the DIO lines, EOI is made true on the last byte to indicate that the message is completed. This is optional, and many instruments send the ASCII characters CR and LF as data instead. Check your instrument's manual.

IFC—Interface Clear. When this line is true, all instruments disconnect to a defined state. (This usually is unaddressed and untalked.) When you turn on the PET, IFC is true for about 100 ms. If the PET is reset, IFC will again be true.

SRQ—Service Request. This permits an instrument to signal that it needs attention . . . and the device in charge of the bus must find out what it needs. The PET has this line as an input, but it takes some programming effort to use SRQ; most instruments don't use SRQ.

REN—Remote Enable. Most IEEE instruments have front panels that permit stand-alone operation—that is, they work as ordinary instruments when the 488 bus isn't connected. REN lets the instrument disconnect from the bus and be controlled from its front panel.

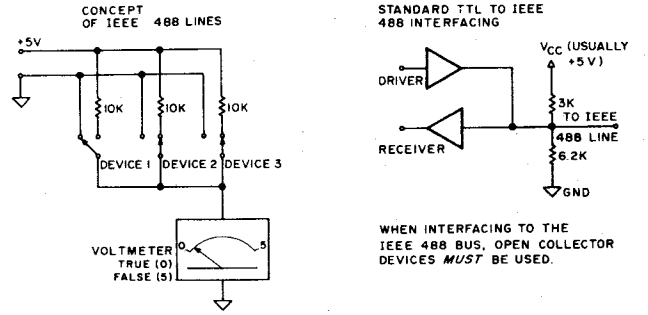


Fig. 4. IEEE 488 equivalent circuits. The lower circuit is the standard method of connecting TTL logic to the 488 bus. The driver must be an open collector and able to sink at least 48 mA at .4 volts and source 5.2 mA at 2.4 or more volts. The PET uses MC 3446P bidirectional line interface ICs for this function.

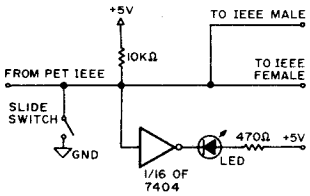


Fig. 5. IEEE "Blinkin Lites" circuit. Each IEEE line uses one copy of this circuit.

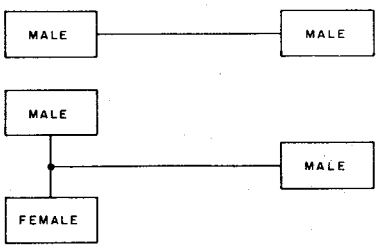


Fig. 3. Convenient cable configurations for the IEEE 488 bus.

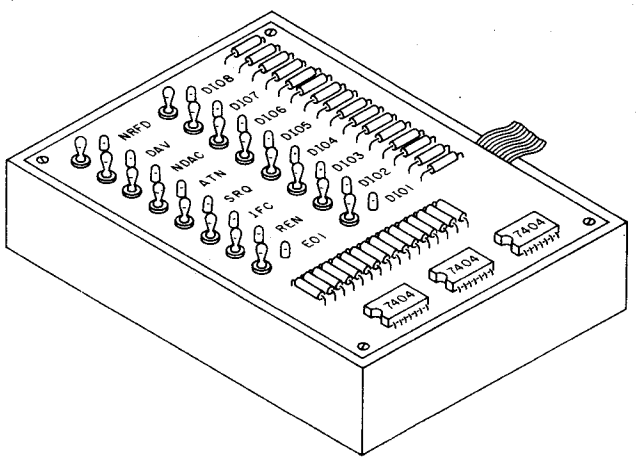


Fig. 6. Sketch of the "Blinkin Lites."

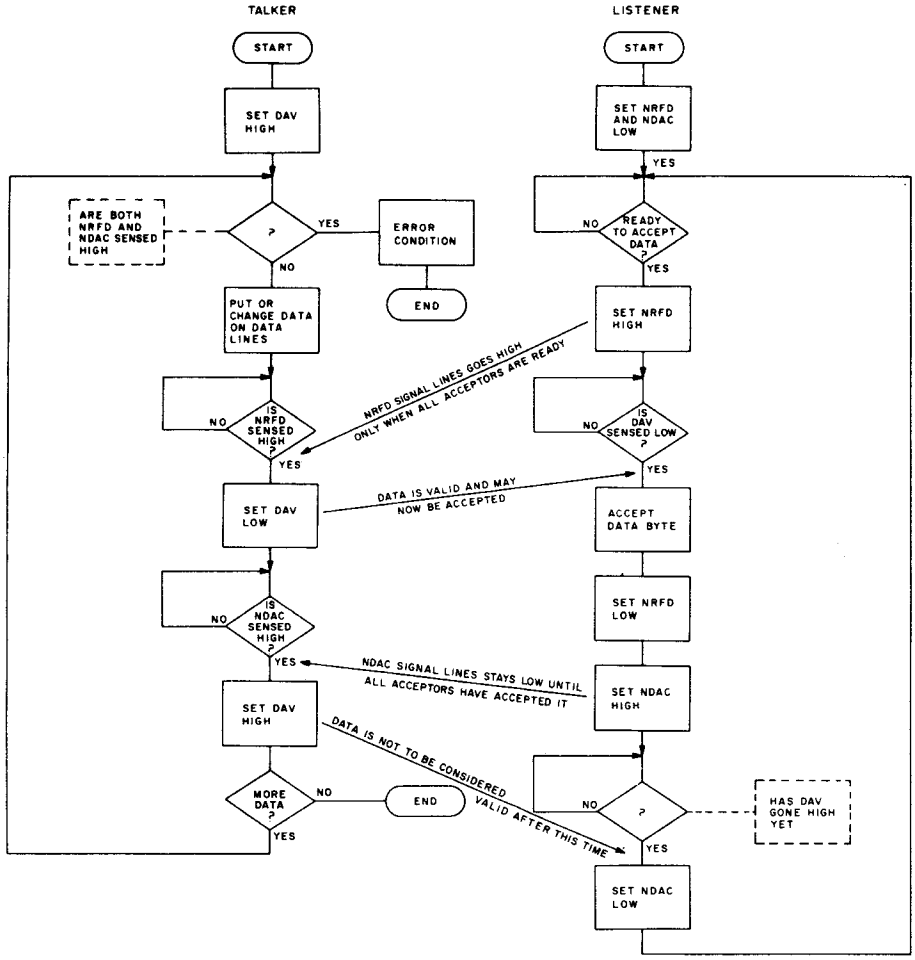


Fig. 7. The IEEE 488 handshake reproduced from Electronics, *Nov. 14, 1974, p. 98, as reproduced in HP part #5952-0058.*

The PET's REN line is always grounded.

ATN—Attention. This is the most relevant line for this article. It tells the device whether to regard the data on the DIO lines as a command or as data. When ATN is true, the byte on the DIO lines is a command. When ATN is false, DIO is seen as data.

The three handshake lines are used to pass bytes on the DIO lines. When a byte is transferred, the slow devices will keep one or more of the handshake lines true until they are finished. This ensures that data is passed at the speed of the slowest device and isn't lost. The handshake lines are:

DAV (Data Valid)—When this line is true, the data on the DIO lines is correct and the receiving instruments can pick up the byte.

NRFD (Not Ready For Data)—When a receiving device is busy or is still processing prior data, it will make NRFD true, which stops data transfers.

NDAC (Not Data Accepted)—When the data is on the DIO lines, the receiving devices keep NDAC true until all of them have read the data byte. Note that the handshake lines don't care whether the data is a command or not; every byte of data or command has to undergo the handshake sequence.

The Handshake

For data transfer, one device is the "talker," which provides the data or commands for transfer. The recipients, or "listeners," pick up the data, and more than one device may listen at the same time. The handshake specifies exactly how the data transfer is accomplished.

Fig. 7 shows a flowchart of the handshake sequence. When the first event, NRFD, goes false, this tells the talker that all of the listeners are now ready to receive a new data byte. The slowest listener is the last one to release NRFD, which will go high.

Next, the talker puts the data byte on the DIO lines and waits briefly to let the signals settle (usually about 10 μs). Once the data is on the DIO lines, NRFD is checked by the talker; if it is false, the talker sets DAV to true. The listeners now know that the new data is ready for pickup. (If NRFD is true, the talker waits until it goes false.)

The first listener that detects DAV true now sets NRFD true, and all of the listeners pick up the data byte from the DIO lines. Up to now, NDAC has been true, and as each listener gets its byte, it releases NDAC. NDAC goes false when all the listeners have the data. The talker waits for NDAC to go false, and when it does, the talker sets DAV to false. The listeners then make NDAC true, and the entire handshake sequence begins again.

Since a device is either a listener, talker or not addressed, Fig. 7 is broken into two flowcharts: one for the talker and one for the listener. A listener will start the handshake with NRFD and NDAC true, while the talker checks these. If both are false—the listener isn't there—an error condition exists.

Commands and Messages

When ATN is true, any data on DIO is seen as a command. Fig. 8 shows the entire ASCII set of 128 characters devoted to IEEE 488 commands.

The ASCII codes 32 through 62 (all numbers in decimal) designate the listen address for a device. Most IEEE-488-compatible devices have a five-position DIP switch next to the 488 connector set to the device's address, a number from 0 to 31. (Note: For the PET, use 4–15.) When the listen address is sent with ATN true and this address matches the device's address, the device will now be addressed to listen and will accept any data sent with ATN false.

If the device is supposed to send data, the talk address—from ASCII codes 64 through 94—will be used instead. The device (if with matching address) will now send data bytes to the bus.

If the device's address (by the switches) is number 7, the listen address value will be 32 + 7, or 39 (apostrophe). The talk address will be 64 + 7, or 71 (letter G). Notice that bits 5-7 designate talk or listen, and bits 0-5 designate the address. Address 31 is reserved for two special commands. Although you can set the switches on a device to 31, it won't operate with this setting.

One instrument must provide these talk and listen addresses. This device is the controller, and the PET is always the controller. The controller can talk and listen too, but only the controller can set ATN true.

Two of the ASCII codes, 63 and 95, serve as "universal" commands. The 63 code is known as "unlisten" and tells all addressed devices to stop listening to the bus. This is faster than trying to tell the devices one at a time to stop listening. The 95 code, "untalk," stops all data transmitters (talkers).

When a message—or a group of data bytes—is sent on the data is present. (In normal operation of the bus, the controller doesn't have to take these drastic measures.)

In some cases, a device will have a secondary address, which permits more than 31 effective addresses on the bus. For example, the Commodore printer might be set as device 4. To control internal functions, secondary addresses select the function in use. (See Commodore's "PET Communication with the Outside World," p. 19.) If a secondary address is in use, it is sent immediately after the talk or listen address, known as the primary address, with ATN true.

Table 1. All PET I/O lines.

IEEE 488 PIA (6520)		ADDRESS: $ E820 59424	
PAØ	IEEE Data In 1	PBØ	IEEE Data Out 1
PA1	" " 2	PB1	" " 2
PA2	" " 3	PB2	" " 3
PA3	" " 4	PB3	" " 4
PA4	" " 5	PB4	" " 5
PA5	" " 6	PB5	" " 6
PA6	" " 7	PB6	" " 7
PA7	" " 8	PB7	" " 8
CA1	ATN In	CB1	SRQ In
CA2	NDAC Out	CB2	DAV Out

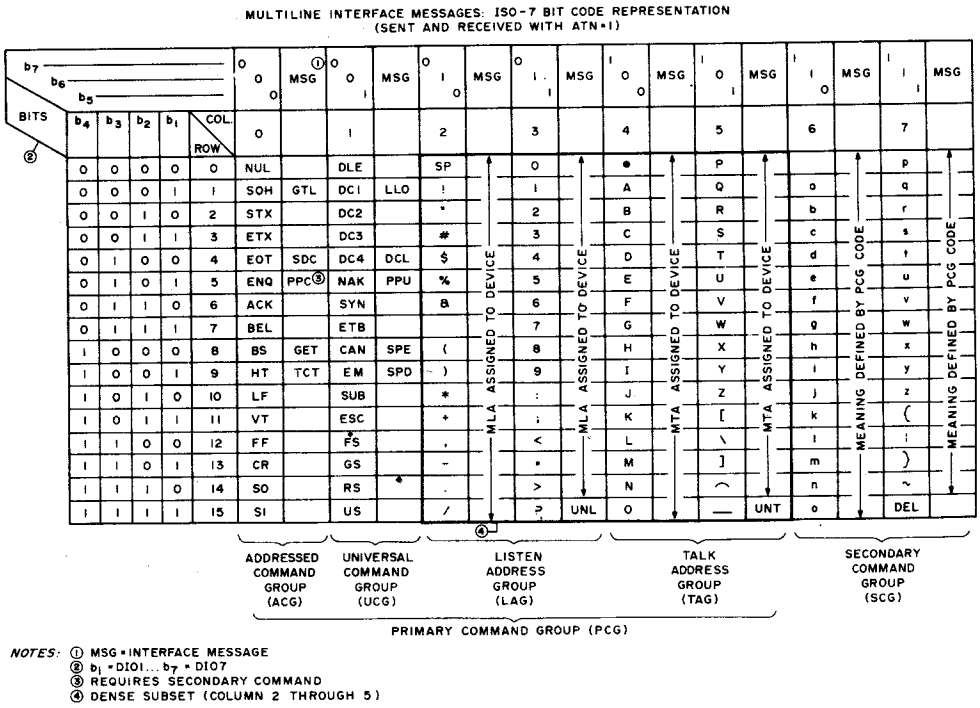


Fig. 8. IEEE 488 command set reproduced from the IEEE Standard 488-1975/ANSI MC 1.1-1975, *p. 77.*

Several of the bus-management lines, such as SRQ, EOI, REN and IFC, serve special functions. Many instruments do respond to these, and often the response depends upon the instrument.

When ATN is low, about half the ASCII code is devoted to special commands, which come in defined sequences whose definition takes about two-thirds of the formal IEEE 488 specification. Most instruments use only a few of these.

488 bus, the controller sets ATN true and sends a listen address; the controller sets ATN true and sends a talk address; the talker puts data on the bus, and the listener picks it up. When the talker is finished, it may set EOI true on the last byte or send CR LF as the last bytes. The controller now sets ATN true and sends untalk (UNT) and unlisten (UNL), which reset the two devices.

In many cases, the controller—in this case, the PET—does the talking or listening. The controller can make everything stop by either setting IFC true or setting ATN true and putting UNT on the bus. Since UNT has its five lowest significant bits true, the active low operation of the IEEE lines overrides whatever

Flipping Bits

The PET ultimately communicates to the rest of the world by the screen and some interface chips—two 6520s and one 6522. (For the specs on these chips, contact MOS Technology.) The 6520 and 6522 chips can only drive one TTL load, so the PET's IEEE lines are connected to some buffer chips to provide the currents needed in the IEEE 488 bus.

Table 1 indicates all of the PET's I/O line assignments as a reference. The PET utilizes all 60 I/O lines as shown here. Most of the IEEE lines are buffered with MC 3446P bidirectional line **driver chips to provide the IEEE current requirements. SRQ is an input only and connects directly to the 6520 chip. IFC is buffered** with a NAND and some resistors to the IEEE specification.

Table 1 reveals some interesting irregularities concerning the IEEE 488 bus: If EOI is true, the PET's display is turned off. (Programs that PEEK and POKE the display area in memory can use this to avoid snow.) Later-model PETs don't have this problem. REN isn't listed; the PET's REN line is wired to ground (true). IFC is not shown. The PET's IFC is connected to the power-on one-shot, which sets IFC true for about 100 ms when the PET is turned on. If you reset the PET by grounding the $\overline{\text{RES}}$ line, IFC may not go true. A better approach is to trigger the power-on one-shot by inserting a switch between power and the 555's power pin. The SRQ line is an input only. The PET's firmware does not use SRQ, so you have to program it directly.

In a 650x-based system, all I/O is seen as a set of memory addresses. This means that BASIC's PEEK and POKE can be used to control the IEEE 488 lines. Table 2 indicates the addresses and bits involved for the PET's IEEE lines. In most cases, a direct PEEK or POKE will do. Two lines, ATN in and SRQ in, require a more complex sequence. These are connected to CA1 and CB1 of a 6520, which set flag bits in the Interrupt Flag register. Resetting these bits requires a memory access to the DIO data register.

Table 3 lists the specific PEEKs and POKEs to individually sense or modify the IEEE lines. In many cases the PEEK or POKE values can be ANDed or ORed together to do several operations at once. If you have built the IEEE Blinkin Lites, try a

```
KEYBOARD PIA (6520)                ADDRESS: $ E810     59408

PAØ  Keyboard Row Select, LSB      PBØ  Keyboard Column A
PA1     "      "   "               PB1     "       "    B
PA2     "      "   "               PB2     "       "    C
PA3     "      "   "    , MSB      PB3     "       "    D
PA4  Switch, Cassette #1           PB4     "       "    E
PA5     "        "    #2           PB5     "       "    F
PA6  EOI In                        PB6     "       "    G
PA7  Diagnostic Jumper             PB7     "       "    H

CA1  Read, Cassette #1             CB1  Video Horiz Sync In
CA2  Screen Blank & EOI Out        CB2  Motor, Cassette #1

The Diagnostic LED will light if PAØ-High, PA1-High, PA2-Low, PA3-High.
```

```
USER PORT VIA (6522)               ADDRESS: $ E840     59456

PAØ  User Port LSB                 PBØ  NDAC In
PA1    "    "                      PB1  NRFD Out
PA2    "    "                      PB2  ATN Out
PA3    "    "                      PB3  Write, Both Cassettes
PA4    "    "                      PB4  Motor, Cassette # 2
PA5    "    "                      PB5  Video Horiz Sync In
PA6    "    "                      PB6  NRFD In
PA7    "    "  MSB                 PB7  DAV In

CA1  User Port Handshake           CB1  Read, Cassette #2
CA2  Characters ROM Select         CB2  User Port Handshake

CA2 selects the MSB of the characters ROM, selecting the PET's
graphics or lower case characters for the display.
```

few of these PEEKs and POKEs to see how they work.

When I was flipping bits with PEEK and POKE for the IEEE lines, I was confused each time I had to figure out the decimal numbers for each changed bit. Perhaps it would be easier to display a byte of memory on the PET's screen in a "front panel" format with simulated LEDs for each bit and some simple keyboard commands to change bits and addresses. Memory Monitor (see Listing 1) does this.

When Memory Monitor is loaded and run, and the first page of instructions is read, the display in Fig. 9 is shown. A box with four parts appears in the middle of the screen with the title Memory Monitor placed above the box. Left of the box is a marker, >>; which indicates the part of the box accessible by the keyboard.

The top of the box shows the address of a memory location in decimal. If you press SPACE, the address will be erased, and a new number can be entered. Pressing number keys enters a new address, and a reverse-field cursor appears.

When a cursor isn't on the screen, pressing RETURN will move the marker to the next part of the box. (The second part in the box indicates the bit numbers and is skipped by the marker.)

The third part of the box displays a front panel made of solid or hollow "balls" (or "LEDs"). This shows the eight bits of the byte under investigation. The numbers above the "LEDs" indicate the bit numbers, 7 the MSB and 0 the LSB. To change the byte, enter 0 or 1 (or Shift-Q and Shift-W), and the cursor will appear. Pressing RETURN enters the value.

The fourth part of the box is the value of the byte in decimal and is entered in the same way

Listing 1. Memory Monitor.

```
10 PRINT"clr sp sp sp sp sp sp sp --> MEMORY MONITOR <--
20 PRINT"dn sp sp THIS PGM DISPLAYS A LOCATION IN THE
30 PRINT"PET'S MEMORY IN BOTH DECIMAL AND IN A
40 PRINT"'FRONT PANEL' FORMAT.
50 PRINT"dn sp sp YOU CAN CHANGE THE ADDRESS OR VALUE
60 PRINT"BY ENTERING A NEW VALUE WHEN THE '>>'
70 PRINT"MARKER IS NEXT TO THE ITEM YOU ARE
80 PRINT"CHANGING.
90 PRINT"dn sp sp PRESS 'RETURN' TO ENTER THE CHANGE
100 PRINT"OR TO MOVE THE MARKER.
110 PRINT"dn sp sp THE PGM CONSTANTLY PEEKS THE LOCATION
120 PRINT"WHEN YOU AREN'T CHANGING A VALUE. IF
130 PRINT"YOU CHANGE THE ADDRESS, THE PGM
140 PRINT"WILL SHOW THE NEW VALUE. IF YOU CHANGE
150 PRINT"A VALUE, IT IS POKED INTO MEMORY.
160 PRINT"dn sp sp 'H' WILL GIVE YOU SOME HELP FOR EACH
170 PRINT"ITEM.
190 PRINT"dn PRESS ANY KEY TO START
195 GETA$:IFA$=""THEN195

200 REM DRAW DISPLAY FORMAT
210 PRINT"clr dn dn dn dn dn";
220 D1$="@ @ @ @ @ @ @ @ @ @ @ @ @ @ @"      (15 shift-@)
230 D3$="rt rt rt rt rt rt rt rt rt rt rt rt"      (10 rt's)
240 PRINT"rt rt rt rt rt rt rt rt --> sp MEMORY MONITOR sp< --"
250 PRINTD3$"Ø"D1$"."
255 PRINTD3$"┃ ADDRESS:sp sp sp sp sp sp sp sp ┃"   (┃ is a vertical line)
260 PRINTD3$"+"D1$"3"
265 PRINTD3$"┃ 7 sp 6 sp 5 sp 4 sp 3 sp 2 sp 1 sp Ø ┃"
270 PRINTD3$"+ @ 2 @ 2 @ 2 @ 2 @ 2 @ 2 @ 3"
280 PRINTD3$"┃ sp ┃ sp ┃ sp ┃ sp ┃ sp ┃ sp ┃ sp ┃ sp ┃"
290 PRINTD3$"+ @ 2 @ 2 @ 2 @ 2 @ 2 @ 2 @ 3"
300 PRINTD3$"┃ DECIMAL: sp sp sp sp sp sp sp ┃"
310 PRINTD3$"-"D1$"="
320 PRINT"dn dn dn"
```

NOTE: For Lines 200-320 see Fig. 9.

```
400 REM IDLING PROGRAM
410 AD=59471:PT=1
500 REM DISPLAY ADDRESS
510 GOSUB 1000
520 REM DISP PANEL LITES
525 DT=PEEK(AD)
530 GOSUB 1200
540 REM DISP DECIMAL
550 GOSUB 1400
560 REM DISP PTR
570 GOSUB 1600
580 REM GET CHAR
590 GET A$
600 IF A$="" THEN 500
610 IF A$=CHR$(13) THEN 700
620 FG=Ø:GOSUB 2500
630 IF FG=Ø THEN 510
640 GOTO 210
700 REM BUMP PTR
710 GOSUB 1800
720 GOTO 510

1000 REM DISP ADDR
1010 PRINT"hm dn dn dn dn dn dn dn"
1020 V$=STR$(AD)+"sp sp sp sp sp sp sp"
1030 V$=MID$(V$,2,6)
1040 PRINT TAB(2Ø);V$
1050 RETURN

1200 REM DISP PANEL
1210 PRINT"dn dn dn"TAB(11);
1220 VT=DT:DV=128
1230 FOR J=1 TO 8
1240 IF VT/DV< 1 THEN 1260
1250 PRINT" Q rt";:VT=VT-DV:GOTO 1300
1260 PRINT" W rt";
1300 DV=DV/2
1310 NEXT J

1400 REM DISP DECIMAL
1410 PRINT"dn"
1420 V$=STR$(DT)+" sp sp sp sp sp sp sp sp "
1430 V$=MID$(V4,2,6)
1440 PRINT TAB(2Ø);V$
1450 RETURN

1600 REM DISP PTR
1610 PRINT"hm dn dn dn dn dn dn dn"
1620 IF PT >1 THEN 1640
1630 PRINT TAB(8)">>";:RETURN
1640 PRINT"dn dn dn"
1650 IF PT >2 THEN 1670
1660 GOTO 1630
1670 PRINT"dn":GOTO 1630

1800 REM BUMP PTR
1810 GOSUB 1600
1820 PRINT"rt rt sp sp"
1830 PT=PT+1: IF PT >3 THEN PT=1
1840 GOSUB 1600
1850 RETURN

2000 RETURN                (This line probably isn't needed)

2500 REM CHANGE ITEM
2510 ON PT GOSUB 3000,3500,4000
2520 RETURN

3000 REM CHANGE ADDR
3010 IF A$="H" THEN GOSUB 4500:RETURN
```

VALUES FOR INPUTS

IEEE LINE	ADDRESS (HEX)	ADDRESS (DECIMAL)	BIT
DIO 1	E820	59424	0
DIO 2	E820	59424	1
DIO 3	E820	59424	2
DIO 4	E820	59424	3
DIO 5	E820	59424	4
DIO 6	E820	59424	5
DIO 7	E820	59424	6
DIO 8	E820	59424	7
EOI	E810	59408	6
IFC	----	-----	-
SRQ	E823	59427	7
REN	----	-----	-
ATN	E821	59425	7
DAV	E840	59456	7
NRFD	E840	59456	6
NDAC	E840	59456	0

VALUES FOR OUTPUTS

IEEE LINE	ADDRESS (HEX)	ADDRESS (DECIMAL)	BIT
DIO 1	E822	59426	0
DIO 2	E822	59426	1
DIO 3	E822	59426	2
DIO 4	E822	59426	3
DIO 5	E822	59426	4
DIO 6	E822	59426	5
DIO 7	E822	59426	6
DIO 8	E822	59426	7
EOI	E811	59409	3
IFC	----	-----	-
SRQ	----	-----	-
REN	----	-----	-
ATN	E840	59456	2
DAV	E823	59427	3
NRFD	E840	59456	1
NDAC	E821	59425	3

Table. 2 Addresses and bits for the IEEE 488 lines.

as the address.

If you press RETURN several times, the marker rotates through the three accessible parts of the box. To recall how to enter a value, press the letter H, which clears the screen and provides instructions.

The Memory Monitor eased the tedium and frustration of checking the PEEKs and POKEs used in the IEEE 488 memory locations. I have made Memory Monitor simple to use, and I consider it a good example of user-oriented programming.

Doing It the Hard Way

With direct access to the PET's IEEE 488 lines, you can use PEEK and POKE to operate an IEEE instrument "by hand." This is probably more difficult than using the IEEE Blinkin .ites box to communicate switch by switch because it takes more keystrokes to change a bit with POKE.

The next step is to write a BASIC program that performs the required IEEE 488 operations directly. Though the PET has these "built in," there are a few advantages to doing the whole thing in BASIC.

Everything goes slowly. As events happen, there is a chance of seeing them as they go by.

BASIC is accessible. If the PET or your instrument decides that the sky's the limit, pressing the STOP key can illuminate where the difficulties lie. The PET's built-in IEEE 488 services

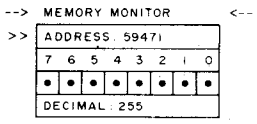


Fig. 9. Listing 1's initial display.

```
3020 V1=AD
3030 GOSUB 5000
3040 IF V2< Ø THEN RETURN
3050 IF V2>65535 THEN RETURN
3060 AD=V2:RETURN

3500 REM CHANGE BINARY VALUE
3510 IF A$="H"THEN GOSUB 4600: RETURN
3520 V1=DT
3530 GOSUB 5500
3540 IF V2< Ø THEN RETURN
3550 IF V2>255 THEN RETURN
3560 DT=V2:POKE AD,DT:RETURN

4000 REM CHANGE VALUE
4010 IF A$="H" THEN GOSUB 4500: RETURN
4020 V1=DT
4030 GOSUB 5000
4040 IF V2< Ø THEN RETURN
4050 IF V2>255 THEN RETURN
4060 DT=V2:POKE AD,DT: RETURN

4500 PRINT"clr sp sp TYPE IN THE NEW NUMBER AND PRESS
4505 FG=1
4510 PRINT"RETURN. PRESS 'X' TO ABORT & NOT MAKE
4515 PRINT"THE CHANGE.
4520 PRINT" sp sp PRESS SPACE TO ERASE REST OF NUMBER.
4530 PRINT"dn sp sp PRESS ANY KEY
4540 GETA$: IFA$=""THEN 4540
4550 RETURN

4600 PRINT"clr sp sp ENTER '1' OR ' Q ' TO SET A BIT, AND
4610 PRINT"'Ø' OR ' W ' TO RESET A BIT. PRESS
4620 PRINT"RETURN WHEN DOEN.
4625 PRINT" sp sp PRESS SPACE TO SKIP A BIT.
4630 PRINT"dn sp sp PRESS ANY KEY
4640 GETA$:IF A$="" THEN 4540
4650 RETURN

5000 REM NUMERIC ENTRY
5010 REM POS CURSOR
5020 PRINT TAB(20);
5030 REM MAKE DISP STR
5040 D$=MID$(STR$(V1),2)+"sp sp sp sp sp sp sp"
5050 D$=LEFT$(D$,6)
5060 REM SET RVS PTR & RETURN VALUE
5070 PC=1:V2=-1
5080 REM SEE INPUT & ACT
5090 IF A$="X" THEN RETURN
5100 IF A$=CHR$(13) THEN V2=VAL(D$):RETURN
5110 IF A$<>" sp " THEN 5120
5112 IF PC=1 THEN D$="sp sp sp sp sp sp":GOTO 5210
5114 D$=LEFT$(D4,PC-1)+"sp sp sp sp sp sp":D$=LEFT$(D$,6)
5118 GOTO 5210
5120 IF A$< "Ø" OR A$>"9" THEN 5210
5125 REM REMAKE STRING
5130 DX$=D$:D$=""
5140 FOR J=1 TO 6
5150 IF PC=J THEN D$=D$+A$:GOTO 5170
5160 D$=D$+MID$(DX$,J,1)
5170 NEXT J
5180 PC=PC+1: IF PC>7 THEN PC=1

5200 REM DISPLAY RESULT & RESTORE CURSOR
5210 FOR J=1 TO 6
5220 IF J=PC THEN PRINT "rvs";
5230 PRINT MID$(D$,J,1);
5240 IF J=PC THEN PRINT "off";
5250 NEXT J:PRINT"lft lft lft lft lft lft";
5260 GET A$: IF A$="" THEN 5260
5270 GOTO 5090

5500 REM BINARY ENTRY
5510 PRINT TAB(11);
5520 FOR J= 1 TO 8
5525 V1=V1/2
5530 IF V1=INT(V1) THEN D$=" W "+D$: GOTO 5540
5535 D$=" Q "+D$
5540 V1 = INT(V1)
5550 NEXT J
5570 REM SET RVS PTR
5580 PC=1:V2=-1
5590 REM LOOK AT INPUT
5600 IF A$="X" THEN RETURN
5605 IF A$=CHR$(13)THEN 5780
5610 IF A$="sp" THEN 5715
5620 IF A$="1" OR A$=" Q " THEN A$=" Q ":GOTO 5660
5630 IF A$="Ø" OR A$=" W " THEN A$+" W ":GOTO 5660
5640 GETA$:IFA$=""THEN 5640
5650 GOTO 5600
5660 REM REMAKE STRING
5670 DX$=D$:D$=""
5680 FOR J= 1 TO 8
5690 IF PC=J THEN D$=D$+A$: GOTO 5710
5700 D$=D$+MID$(DX$,J,1)
5710 NEXT J
5715 PC=PC+1:IF PC>8 THEN PC=1
5720 REM DISP & FIX CURSOR
5730 FOR J= 1 TO 8
5735 IF J = PC THEN PRINT "rvs";
5740 PRINT MID$(D$,J,1)"rt";
5745 IF J=PC THEN PRINT "off";
5750 NEXT J:PRINT" lft lft lft lft lft lft lft lft
                lft lft lft lft lft lft lft lft";  (16 lft's)
5760 GOTO 5640
5770 REM MAKE VALUE
5780 V2=Ø:FORJ=1 TO 8
5785 V2=V2*2
5790 IF MID$(D$,J,1)=" W " THEN 5810
5800 V2=V2+1
5810 NEXT J
5820 RETURN
```

are mostly invisible, and there's often no way to find out why something went wrong.

Everything is under control. It is simple enough to display every step with suitable messages to the screen. If necessary, you can insert a GET loop to make the PET wait until a key is pressed before proceeding.

Changes are easy.

It's an educational experience—those who must learn the "nuts and bolts" of the IEEE bus will find a BASIC emulator useful.

I constructed the BASIC 488 program (see Listing 2) to provide the following essential services: put the PEEK and POKE values into variable form for reasonably fast execution and to simplify debugging with direct commands; do most of the PEEKs and POKEs for line control as short subroutines; provide the listen and talk handshake sequences for one byte and display their progress; provide a way to send and receive strings to a device on the bus; set the program up as a skeleton onto which you can add specific programs to suit changing needs.

Table 4 indicates the subroutines and variables used in the BASIC 488 program. Load these subroutines and then add the code you need for your devices. Some devices, such as those by Commodore, may not follow the IEEE time standard, and the BASIC 488 program will not be fast enough to prevent time-outs.

I built the program from the bottom up, starting with subroutines 1500 and the series starting at 9000. Subroutine 1500 sets up the essential variables. A1-7 are the addresses of the PEEK/POKE locations; M0-M7 and N0-N8 are AND and OR masks to extract bits 0-7 from a location (or to set the desired bits); 01-07 are the original values for addresses A1-A7. (POKE A1,01, for example, will restore location A1 to the PET's power-on value, which helps you to recover from disasters.)

The variables H1 to H6 are the sense values for the IEEE lines. For example, if H1 is 1, the DAV line is true. If H1 is zero, DAV is false.

When you enter BASIC 488, enter lines 1000-1620 and lines 9000-9640 first. Use the IEEE Blinkin Lites to check that the subroutines in the 9000 series function correctly. First, GOSUB 1000 in direct mode to set things up. Then, GOSUB to the section under test and look at the Blinkin Lites to see what happened. A PRINT H1 will inform you of the sensing subroutines' results. Be sure to thoroughly test the 9000 series first!

All DIO Lines:

IN:	POKE 59426,255:V = PEEK(59424):V = (NOT(V))AND255
OUT:	V = (NOT(V))AND255: POKE 59426,V

EOI

IN:	V = 1:IF PEEK(59408)AND 64 THEN V = 0
TRUE OUT:	POKE 59409, PEEK(59409) AND 247
FALSE OUT:	POKE 59409, PEEK(59409) OR 8

REN & IFC—Not Applicable

SRQ**

IN:	V = 0:IF PEEK(59427) AND 128 THEN V = 1
	Z = PEEK(59426)
LO-HI	POKE 59427, PEEK(59427) OR 2
HI-LO	POKE 59427, PEEK(59427) AND 253

ATN**

IN:	V = 0:IF PEEK(59425) AND 64 THEN V = 1
	Z = PEEK(59424)
LO-HI:	POKE 59409, PEEK(59409) OR 2
HI-LO:	POKE 59409, PEEK(59409) AND 253
TRUE OUT:	POKE 59456, PEEK(59456) AND 251
FALSE OUT:	POKE 59456, PEEK(59456) OR 4

DAV

IN:	V = 1: IF PEEK(59456) AND 128 THEN V = 0
TRUE OUT:	POKE 59427, PEEK(59427) AND 247
FALSE OUT:	POKE 59427, PEEK(59427) OR 8

NRFD

IN:	V = 1: IF PEEK(59456) AND 64 THEN V = 0
TRUE OUT:	POKE 59456, PEEK(59456) AND 253
FALSE OUT:	POKE 59456, PEEK(59456) OR 2

NDAC

IN:	V = 1: IF PEEK(59456) AND 1 THEN V = 0
TRUE OUT:	POKE 59425, PEEK(59425) AND 247
FALSE OUT:	POKE 59425, PEEK(59425) OR 8

*The extra parenthesis in the complementation of V is required, for the PET evaluates AND before NOT.

**The HI-LO or LO-HI determines which transition the CA/CB1 inputs will respond to. Set the HI-LO or LO-HI before doing the IN: line. The Z = PEEK resets the flag bit. Be sure to reset the flag bit before checking the first time.

SRQ OUT: Not available on the PET.

Table 3. PEEKs and POKEs for the IEEE 488 lines.

Listing 2. BASIC 488 program.

```
1000 REM **** IEEE 488 ****
1005 REM   GREGORY YOB, JAN 1979
1010 REM   BOX 354, PALO ALTO CA 94301
1015 REM
1020 REM THESE ROUTINES PERMIT DIRECT
1025 REM MANIPULATION OF THE PET IEEE
1030 REM 488 BUSS LINES AND (SLOW!!)
1035 REM IEEE 488 COMMAND AND DATA
1040 REM TRANSFERS
1045 REM
1500 REM -- INITIALIZATION --
1510 RESTOR:READ A1,A2,A3,A4,A5,A6,A7
1520 DATA 59424,59426,59425,59427,59408,59456,59409
1530 READ M0,M1,M2,M3,M4,M5,M6,M7
1540 DATA 1,2,4,8,16,32,64,128
1550 READ N0,N1,N2,N3,N4,N5,N6,N7
1560 DATA 254,253,251,247,239,223,191,127
1570 READ N8
1580 DATA 255
1590 READ 01,02,03,04,05,06,07     (Each of these is Letter O)
1600 DATA 255,266,60,60,249,255,60
1610 DEF FNF(X)=(NOT(X))AND255
1620 RETURN

7000 PRINT"clr GET MESSAGE"
7010 PRINT"dn PRESS KEY TO START"
7020 GETA$:IFA$=""THEN RETURN        ("" is an empty string)
7030 D2=FNF(DV+64):GOSUB9450:GOSUB8500:GOSUB9470
7040 B$=""
7050 GOSUB 8000:IF FNF(D1)=13THEN7070
7060 B$=B$+CHR$(FNF(D1)):GOTO7050
7070 GOSUB8000:REM LF BUCKET
7080 PRINT"dn dn MESSAGE IS: sp"B$
7090 RETURN

7500 PRINT"clr SEND MESSAGE"
7510 INPUT"dn dn MESSAGE:";C$
7520 D2=FNF(DV+32):GOSUB9450:GOSUB8500:GOSUB9470
7530 FOR J=1 TO LEN(C$)
7540 D2=FNF(ASC(MID$(C$,J)))
7550 GOSUB8500:NEXTJ
7560 PRINT"dn dn MESSAGE SENT: sp"C$
7570 RETURN

8000 PRINT"clr LISTEN HANDSHAKE dn"
8010 GOSUB9350:GOSUB9250:GOSUB9370
8020 PRINT" sp NRFD TRUE dn":PRINT" sp NDAC TRUE"
     :PRINT" sp NRFD FALSE"
8030 PRINT"WAITING FOR DAV TRUE"
8040 GETA$:IFA$<>""THENPRINT"--FORCED":GOTO8060
8050 GOSUB9100:IFH1=0THEN8040
8060 GOSUB9000:PRINT"dn spDATA:"FNF(D1)CHR$(FNF(D1))
8070 GOSUB9350:GOSUB9270
8080 PRINT"dn sp NRFD TRUE":PRINT" sp NDAC FALSE"
8090 PRINT"WAITING FOR DAV FALSE"
8100 GETA$:IFA$<> ""THENPRINT"--FORCED":GOTO8120
8110 GOSUB9100:IFH1=1THEN8100
8120 GOSUB9250
8130 PRINT"dn sp NDAC TRUE"
8140 RETURN

8500 PRINT"clr TALK HANDSHAKE"
8510 GOSUB9170
8520 PRINT"dn sp DAV FALSE"
8530 GOSUB9200:GOSUB9300
8540 IF H1+H2>0 THEN 8570
8550 PRINT"dn>> sp ERROR STATE-PRESS KEY TO FORCE"
8555 PRINT"NOTE: MAKE NRFD, NDAC TRUE"
8560 GETA$;IFA$=""THEN8560
8570 GOSUB9050
8580 PRINT"dn DATA ON LINE:"FNF(D2)CHR$(FNF(D2))
```

Nothing else will work if these don't!

If all else fails, refer to Tables 1, 2 and 3 and try a few direct PEEKs and POKEs to ensure that the IEEE lines are functional.

Add lines 8000-8140 and lines 8500-8690, which you can check by attaching the 488 Blinkin Lites and carefully tracing through the handshake flowchart in Fig. 7. Again, it is essential to be sure these routines work correctly. An additional benefit is that you will learn the handshake sequence in detail.

Note that the data transferred, D1 or D2, must be complemented with the FNF function as it enters or leaves the IEEE bus. In some of the waiting loops, such as lines 8030-8050, a GET A$ check is inserted. If the instrument hangs up, pressing a key will force the handshake to proceed, and a suitable message will appear on the screen. As the handshakes proceed, their progress is reported to the screen for your reference.

Next, add lines 7000-7570. These routines require a device address, DV, to function correctly. Subroutine 7000 will fetch a message from a device, and subroutine 7500 will send a message. The strings B$ and C$ are used to store the messages.

Most devices will send an EOI along with the last character of their messages. This will turn off the screen. In some cases, you will have to provide an EOI, which will again turn off the screen. To recover, enter:

GOSUB 9570 (and RETURN)

Another approach is to move the cursor down until the screen scrolls. A scroll turns the screen off, and then on. If you have a 16K PET, the screen will not blink.

Testing the last part via the IEEE Blinkin Lites is tedious. If you have an instrument available, try talking to it! Be sure you know *exactly* what your instrument expects and its responses!

Talking to the HP Clock via BASIC 488

Now that you have checked out BASIC 488 by hand, try it with a real live instrument! I connected the HP clock, loaded BASIC 488 and gave it a try (see Example 1). The clock's front panel shows the reset worked.

These commands can be compressed to one line (see Example 2).

Next, try to read the clock. Address the clock to talk, then read the 14-character message shown in Example 3. If you look at the line DATA: on the display for the Listen Handshake, you can barely see the clock's message. A different version (see Example 4) will pick up the message and leave it later. Below the Listen Handshake display appears the clock's message:
0101000520

The BASIC 488 program has two routines for sending and reading entire strings via the IEEE 488. Subroutine 7000 addresses device DV to talk and read a string. Subroutine 7500 addresses device DV to listen and sends a string. (Note: Routine 7000 reads a string until a carriage return is seen, and then reads one more character. This is because the HP clock ends messages with CR and LF. You might have to change this for your device.)

To reset the clock:
DV = 7:GOSUB 7500
The screen clears and asks for

```
8590 print"dn WAITING FOR NRFD FALSE"
8600 GETA$:IFA$<>""THENPRINT"--FORCED":GOTO8620
8610 GOSUB9300:IFH3=1THEN8600
8620 GOSUB9150.
8630 PRINT"dn sp DAV TRUE"
8640 PRINT"WAITING FOR NDAC FALSE"
8650 GETA$:IFA$<>""THEN8670
8670 GOSUB9170
8680 PRINT"dn sp DAV FALSE"
8690 RETURN

9000 POKEA2,N8:D1=PEEK(A1):RETURN
9050 POKEA2,D2:RETURN
9100 H1=1:IFPEEK(A6)ANDM7THENH1=Ø
9110 RETURN
9150 POKEA4,PEEK(A4)ANDN3:RETURN
9170 POKEA4,PEEK(A4)ORM3:RETURN
9200 H2=1:IFPEEK(A6)ANDMØTHENH2=Ø
9210 RETURN
9250 POKEA3,PEEK(A3)ANDN3:RETURN
9270 POKEA3,PEEK(A3)ORM3:RETURN
9300 H3=1:IFPEEK(A6)ANDM6THENH3=Ø
9310 RETURN
9350 POKEA6,PEEK(A6)ANDN1:RETURN
9370 POKEA6,PEEK(A6)ORM1:RETURN
9400 PRINT"NO ATN LEVEL":STOP
9430 H4=Ø:IFPEEK(A3)ANDM7THENH4=1
9440 ZZ=PEEK(A1):RETURN
9450 POKEA6,PEEK(A6)ANDN2:RETURN
9470 POKEA6,PEEK(A6)ORM2:RETURN
9500 H5=1:IFPEEK(A5)ANDM6THENH5=Ø
9510 RETURN
9550 POKEA7,PEEK(A7)ANDN3:RETURN
9570 POKEA7,PEEK(A7)ORM3:RETURN
9600 REM SRQ NOT OUTPUT
9630 H6=Ø:IFPEEK(A4)ANDM7THENH6=1
9640 ZZ=PEEK(A2):RETURN
```

Entry Points:

SUBROUTINE 1500	Initialization (Must be done first)
SUBROUTINE 7000	Get Message as B$, Requires DV
SUBROUTINE 7500	Put Message C$, Requires DV
SUBROUTINE 8000	Listen Handshake
SUBROUTINE 8500	Talk Handshake
SUBROUTINES 9000ff9600	IEEE Lines Primitives
9000	Read DIO as D1
9050	Write DIO as D2
9100	Read DAV as H1
9150	Set DAV TRUE
9170	Set DAV FALSE
9200	Read NDAC as H2
9250	Set NDAC TRUE
9270	Set NDAC FALSE
9300	Read NRFD as H3
9350	Set NRFD TRUE
9370	Set NRFD FALSE
9400	Trap for ATN
9430	Check ATN as H4 (If changed)
9450	Set ATN TRUE
9470	Set ATN FALSE
9500	Read EOI as H5
9550	Set EOI TRUE (Screen will blank)
9570	Set EOI FALSE (Screen returns)
9630	Check SRQ as H6 (If changed)

Variables:

PEEK/POKE ADDRESSES		ORIGINAL VALUES	
A1	59424	01	255
A2	59426	02	255
A3	59425	03	60
A4	59427	04	60
A5	59408	05	249
A6	59456	06	255
A7	59409	07	60

Masks:

M0	0000 0001	1	N0	1111 1110	254
M1	0000 0010	2	N1	1111 1101	253
M2	0000 0100	4	N2	1111 1011	251
M3	0000 1000	8	N3	1111 0111	247
M4	0001 0000	16	N4	1110 1111	239
M5	0010 0000	32	N5	1101 1111	223
M6	0100 0000	64	N6	1011 1111	191
M7	1000 0000	128	N7	0111 1111	127
			N8	1111 1111	255

Miscellaneous:

DV	Device Address
A$	Keyboard dummy entry
B$	Message from Device
C$	Message to Device

Functions:

FNF(X)	Returns complement of argument

Table 4. BASIC 488 program notes.

```
CLR
GOSUB 1500:GOSUB 1900        Get everything ready . . .
PRINT FNF(32 + 7)
  216                        This is the value for D2 as a listen address.
GOSUB 9450                   Make ATN true.
D2 = 216:GOSUB 8500          Send listen address via handshake
TALK HANDSHAKE               The PET responds with the step-by-step
  DAV FALSE                  output handshake and goes successfully
DATA ON LINE: 39             through the entire process.
WAITING FOR NRFD FALSE
  DAV TRUE
WAITING FOR NDAC FALSE       The HP Clock's "addressed" light turns on!
  DAV FALSE
READY.
GOSUB 9470                   Make ATN false
PRINT FNF(ASC("R"))          R resets the clock
  173
D2 = 173:GOSUB 8500          Send 'R' as data . . . . .
( . . . . . )                And this handshakes through OK too.
```

Example 1. My dialogue with the HP clock via BASIC 488.

```
D2 = 216:GOSUB9450:GOSUB8500:GOSUB9470:D2 = 173:GOSUB8500
```

Example 2. A one-line command for Example 1.

```
PRINT FNF(64 + 7)                    Find out D2 for talk address
 184
D2 = 184:GOSUB9450:GOSUB8500:GOSUB9470
( . . . . . )                        The handshake goes through
FOR J = 1 TO 14:GOSUB 8000:NEXT
( . . . . . for 14 times . . . . . )
```

Example 3. The dialogue for reading the clock.

the message (see Example 5).

The Talk Handshake flashes on the screen twice, and the message sent is displayed below:

```
( . . . . . )
MESSAGE SENT: R
```

The program uses routine 7000 to read the time. Since DV is already set, we don't have to reassign DV = 7 again. See Example 5. Note that there are three spaces between the colon and the first zero. Two of these are from the HP clock, which starts all messages with two blanks.

The BASIC 488 program, though slow to operate, never times-out and lets you control the IEEE 488 bus. This is helpful when you debug a new IEEE device with your PET.

If you are an experienced 6502 programmer, it is simple to translate the BASIC 488 program into a set of machine-language routines. If you do so, I'd like a copy (tape and source). Listing 3 shows a copy of the IEEE handshakes in machine language. (From the *PET User Notes*, PO Box 371, Montgomeryville, PA 18936, Vol. 1, Issue 7, (Nov.-Dec. '78), p. 8. This is a reprint from the Commodore PET Users Club of England.)

The PET handles the IEEE 488 as a file. Part 2 will cover this. ■

IEEE Bus Handshake Routine - Main Program

```
1800 A200    LDX #00     prepare index register
1802 A9FB    LDA #FB     set ATN low
1804 2D40E8  AND E840
1807 8D40E8  STA E840
180A A928    LDA #28     MLA (28 for this device)
180C 8501    STA 01
180E 208018  JSR 1880    handshake into bus
1811 A908    LDA #08     GET
1813 8501    STA 01
1815 208018  JSR 1880    handshake
1818 A948    LDA #48     MTA
181A 8501    STA 01
181C 208018  JSR 1880    handshake
181F A9FD    LDA #FD     set NRFD low
1821 2D40E8  AND E840       (ready to receive data)
1824 8D40E8  STA E840
1827 A9F7    LDA #F7     and NDAC low also
1829 2D21E8  AND E821
182C 8D21E8  STA E821
182F A904    LDA #04     set ATN high
1831 0D40E8  ORA E840
1834 8D40E8  STA E840
1837 A008    LDY #08     ready to count 8 bytes
1839 208018  JSR 1880    handshake data from bus
183C A502    LDA 02      result to A
183E 9D0119  STA 1901,X  store in 1901+X
1841 E8      INX
1842 88      DEY
1843 D0F4    BNE 1839    jump if Y not zero
1845 A9FB    LDA #FB     set ATN low
1847 2D40E8  AND E840
184A 8D40E8  STA E840
184D A902    LDA #02     set NRFD high
184F 0D40E8  ORA E840
1852 8D40E8  STA E840
1855 A908    LDA #08     set NDAC high
1857 0D21E8  ORA E821
185A 8D21E8  STA E821
185D A95F    LDA #5F     UNT
185F 8501    STA 01
1861 208018  JSR 1880    handshake to bus
1864 A904    LDA #04     set ATN high
1866 0D40E8  ORA E840
1869 8D40E8  STA E840
186C CE0019  DEC 1900    decrease counter
186F D091    BNE 1802    jump if not zero
1871 60      RTS         return to BASIC program
```

Subroutine to Handle Handshake Into Bus

```
1880 AD40E8  LDA E840    NRFD ?
1883 2940    AND #40
1885 F0F9    BEQ 1880    jump back if not ready
1887 A501    LDA 01      ready:  get data byte
1889 49FF    EOR #FF     complement it
188B 8D22E8  STA E822    send to bus
188E A9F7    LDA #F7     set DAV low
1890 2D23E8  AND E823
1893 8D23E8  STA E823
1896 AD40E8  LDA E840    NDAC ?
1899 2901    AND #01
189B F0F9    BEQ 1896    jump back if not accepted
189D A908    LDA #08     accepted;  set DAV high
189F 0D23E8  ORA E823
18A2 8D23E8  STA E823
18A5 A9FF    LDA #FF     255(10) into bus
18A7 8D22E8  STA E822
18AA 60      RTS         return to main
```

Subroutine to Handle Handshake From Bus

```
18B0 A902    LDA #02     set NRFD high
18B2 0D40E8  ORA E840
18B5 8D40E8  STA E840
18B8 AD40E8  LDA E840    DAV ?
18BB 2980    AND #80
18BD D0F9    BNE 18B8    jump back if not valid
18BF AD20E8  LDA E820    get data byte from bus
18C2 49FF    EOR #FF     complement
18C4 8502    STA 02      store in $ 0002
18C6 A9FD    LDA #FD     set NRFD low
18C8 2D40E8  AND E840
18CB 8D40E8  STA E840
18CE A908    LDA #08     set NDAC high
18D0 0D21E8  ORA E821
18D3 8D21E8  STA E821
18D6 AD40E8  LDA E840    DAV high ?
18D9 2980    AND #80
18DB F0F9    BEQ 18D6    jump back if not
18DD A9F7    LDA #F7     set NDAC low
18DF 2D21E8  AND E821
18E2 8D21E8  STA E821
18E5 A9FF    LDA #FF     255(10) into bus
18E7 8D22E8  STA E822
18EA 60      RTS         return to main
```

IEEE Bus Handshake Routine Object Listing

```
1800 A2 00 A9 FB 2D 40 E8 8D
1808 40 E8 A9 28 85 01 20 80
1810 18 A9 08 85 01 20 80 18 A9
1818 A9 48 85 01 20 80 18 A9
1820 FD 2D 40 E8 8D 40 E8 A9
1828 F7 2D 21 E8 8D 21 E8 A9
1830 04 0D 40 E8 8D 40 E8 A0
1838 08 20 B0 18 A5 02 9D 01
1840 19 E8 88 D0 F4 A9 FB 2D
1848 40 E8 8D 40 E8 A9 02 0D
1850 40 E8 8D 40 E8 A9 08 0D
1858 21 E8 8D 21 E8 A9 5F 85
1860 01 20 B0 18 A9 04 0D 40
1868 E8 8D 40 E8 CE 00 19 D0
1870 91 60 EA EA EA EA EA EA
1878 EA EA EA EA EA EA EA EA
1880 AD 40 E8 29 40 F0 F9 A5
1888 01 49 FF 8D 22 E8 A9 F7
1890 2D 23 E8 8D 23 E8 AD 40
1898 E8 29 01 F0 F9 A9 08 0D
18A0 23 E8 8D 23 E8 A9 FF 8D
18A8 22 E8 60 EA EA EA EA EA
18B0 A9 02 0D 40 E8 8D 40 E8
18B8 AD 40 E8 29 80 D0 F9 AD
18C0 20 E8 49 FF 85 02 A9 FD
18C8 2D 40 E8 8D 40 E8 A9 08
18D0 0D 21 E8 8D 21 E8 AD 40
18D8 E8 29 80 F0 F9 A9 F7 2D
18E0 21 E8 8D 21 E8 A9 FF 8D
18E8 22 E8 60

0001 data to go into bus
0002 data from bus

1900 counter for number of data transfers

1901 start of results area
```

Listing 3. IEEE bus handshake routine in machine language. MLA is My Listen Address; MTA is My Talk Address; UNT is Untalk Command.

```
X$ = "":FORJ = 1TO14:GOSUB8000:X$ = X$ + CHR$(FNF(D1)):NEXT:PRINTX$
0101000520
```

Example 4. Putting the clock's message into X$, and the contents of X$.

```
DV = 7:GOSUB7500
SEND MESSAGE

MESSAGE:? R                    R for reset

GET MESSAGE

PRESS KEY TO START

(.....A lot of Listen Handshakes....)

MESSAGE IS:  0101000158
```

Example 5. Resetting the clock.

Program Listing Conventions

The PET's graphics and cursor control characters aren't easily duplicated for program listings, so the conventions described here will be used instead.

If a letter or numeral (or any character) is underlined, it means the corresponding graphics character is to be used. (A is the spade symbol on the PET.)

Lowercase letters indicate PET special functions:

clr	Clear Screen	hm'	Home Cursor
rt	Cursor Right	lft	Cursor Left
up	Cursor Up	dn	Cursor Down
rvs	RVS field on	off	RVS field off
cr	RETURN key	sp	SPACE key

Sp in a line indicates leading or more-than-one blank. For example, dn/sp/sp/HELLO THERE means Cursor Down space HELLO space THERE.

Two IEEE 488 Instruments

The two instruments described here are typical in the way they are controlled via the IEEE 488 bus. Most instruments are controlled by sending and receiving ASCII characters, which are mnemonics of the function being controlled. For example, the HP clock uses the letter D to increment its days' counter. Numbers are usually sent as ASCII strings—in the same way that PRINT provides an ASCII string of digits to a terminal. CR and LF usually indicate a message's end.

Some instruments will use more difficult formats. Two popular forms are BCD, in which two digits per byte are sent, and pure binary, where the value 0-255 is sent. Be sure you know the *exact* formats used by your instruments! Most instruments are unforgiving of bad data; and the responses range from ignoring meaningless characters to the instrument's unaddressing and leaving the bus. Check your instrument's manual!

The HP 59309A Digital Clock

The HP clock is almost the simplest instrument that uses the IEEE 488 bus. Your options are to either set the time or read the time.

When the clock is addressed to talk, it will provide a string of characters with the time in the following format:

(sp or ?) sp NNDDHHMMSS cr lf

The first character is a space or a question mark. If the clock hasn't been set since the last power-off, the question mark will indicate this. The next two digits indicate the month, from 01 to 12. Then comes the day of the month, 01 to 31. (The clock keeps track of the days in each month correctly and has a leap-year switch). Then the hours (00 to 23), minutes and seconds are sent. The carriage return and line feed indicate the end of the message.

Inside the clock are switches that provide variations of the format—colons or commas can either separate the fields, i.e., NN:DD:HH:MM:SS, or simply send the 24-hour time.

When the clock is addressed to listen, eight ASCII characters are used for control:

P—Stop the clock
T—Start the clock
R—Reset the 01:01:00:00:00
S—Each S will increment the Seconds counter
M—Increment Minutes counter
H—Increment Hours counter
D—Increment Days counter
C—Note time, send it when addressed to talk.

For example, the following string will reset the clock to Jan 5, 8:07:12 AM.

PRDDDDHHHHHHHHMMMMMMSSSSSSSSSSST

The T at the end restarts the clock.

The HP 8165A Programmable Signal Source

This is a "cadillac" 488 instrument—the front panel of this machine has 41 buttons for selection of modes and a 12-button number pad for entering times, and frequencies. This works out to 35 different command formats for setting up parameters and switch settings and nine commands for telling the controller the machine's setting or starting a sequence of actions. Some of the formats include:

F1—Select Sine Wave
F2—Select Triangle Wave
F3—Select Square Wave
FRQ f MZ—Select frequency in MHz. f is a number from 1 to 9999.
FRQ f MZ—Same for Hz
FRQ f KHZ—Same for kHz
SET:—Report all parameters currently operating when addressed to talk.
SET: n—Report setting in memory # n (0-9)

The 8165 can store up to ten complete settings in its memories, so the SET commands permit the controller to find out what's in the 8165.

An instrument of this complexity is usually programmed with a set of special-purpose programs as needed. Writing a general-purpose BASIC program would be both tedious and wasteful. My experience is that the hardest part is to get the PET and the instrument to communicate. Once that is accomplished, the rest is easy.

Gregory Yob
Box 354
Palo Alto, CA 94301

Get Your PET on the IEEE 488 Bus

Part 2 of this "opus computerus" examines the file characteristics of the IEEE 488 bus.

Your PET has a "built-in" way of communicating through the IEEE 488 bus. In BASIC, the IEEE 488 looks like a file—just as the cassettes are files. The OPEN statement is used to specify a physical device number of 4 to 30, and the open logical file now talks via the IEEE 488 bus.

A complete understanding of PET tape files is a prerequisite for working with the IEEE 488 as a BASIC file. An article in the January 1979 *Kilobaud Microcomputing* ("PET Techniques Explained") covers many "innocent" errors that will result in mysterious malfunctions.

IEEE 488 Information Transfers

Talking to a Device.

1. OPEN a BASIC file to the device's address. For example, OPEN 1,4 will open the IEEE bus to device 4. Your BASIC program will see this as file #1.

2. PRINT# to your OPENed file. PRINT#1,"HELLO, DEVICE" will address the device to listen, send the string HELLO, DEVICE, add a carriage return with EOI true and then issue the UNT (Untalk) command.

3. Repeat step 2 as needed. Note that after each PRINT#, the IEEE bus is free, since the UNT has been sent.

PRINT# will send the same characters, including the skip character after numbers, as PRINT does to the screen. If you want to send several items, be sure that any needed delimiters, such as ",", are included.

Listening to a Device.

1. OPEN a BASIC file to the device's address.

2. Use INPUT# or GET# to fetch a line or a character from the IEEE bus.

3. Check the status word, ST, for an error, such as time-out. If the device is slow, the PET will complete the INPUT# or GET# and put a nonzero value into ST, which must be checked immediately after the I/O operation. If ST indicates a time-out, jump back to step 2.

4. Convert the data from the INPUT# or GET# as needed, and if more is needed, go to step 2.

Note that after each INPUT# or GET#, the UNT command is sent to the IEEE bus. This will truncate long messages from the device, especially with GET#. Also note that INPUT# (string) and GET# (string) work the best. The BASIC string functions (MID$, RIGHT$, LEFT$ and VAL) will help you get the data into a usable form.

Talking to More than One Device.

1. OPEN a file for each device.

2. Using CMD, send a dummy message to each device. For example, CMD 1:CMD 2:CMD 3 will set up each device (as specified in the OPENs for files 1, 2 and 3) by sending carriage returns to the devices and leaving them as listeners on the bus.

3. PRINT# to the IEEE bus. Any of the OPENed files may be used.

4. Repeat steps 2 and 3 as needed. Since PRINT# ends with the UNT, step 2 must be repeated after each PRINT#.

Transfer from One Device to Another.

1. OPEN a file for each device.

2. CMD to the device that is to be the listener.

3. INPUT# from the device that is to be the talker.

4. Repeat step 3 as needed.

INPUT# does not send a UNL, so the device that was CMDed remains on the bus as a listener. All information sent by the talker to the PET is also received by the listener. To turn off the listener, use a PRINT# to the listener's file. If the talker is slow, check ST and repeat step 3 as required.

LISTing a BASIC Program to a Device

1. OPEN a file to the device.

2. CMD to the device.

3. Enter the LIST command.

4. When the LIST is finished, do a CLR.

The PET's graphics and cursor characters will not print correctly on a standard ASCII printer. (I have a BASIC listing program available.)

The best way to learn the PET files and IEEE 488 is by specific examples. After a detour through CMD, we will continue with two examples. These should provide you with enough information to get started. If you have no success, refer to the section on Common Errors (found later in this installment).

CMD

CMD is an unusual PET command. Consider its functions:

1. Anything that BASIC wants to say is now routed to the device that CMD's file number refers to. If this isn't the screen, nothing that BASIC says will appear on the screen.

2. If a list of variables and literals is provided after the CMD, they will be sent to the device in the same way as PRINT# will.

3. However, if the device is on the IEEE bus, no UNL will be sent, so the device will remain in the listening state and receive any following data sent on the IEEE bus.

To see how CMD operates, get two scratch tapes and enter the program in Example 1. Now SAVE and VERIFY this program on one of your tapes. Put the other tape in the tape unit and execute the following:

```
OPEN 1,1,1
PRESS PLAY & RECORD ON TAPE#1
```

Perform this and wait until the tape stops.

```
OK
READY.
```

Now enter CMD 1. Note that READY. didn't appear; it was provided by BASIC and is now residing in the tape buffer. The cursor is blinking below the C in CMD. Continue with:

```
10  REM CMD EXAMPLE
20  PRINT"***"
30  OPEN 1
40  GET#1, A$
50  PRINT A$;
60  IF A$ = CHR$(90)THEN PRINT"***":END
70  GOTO 40
80  REM Z
```

Example 1.

```
LIST
CLOSE 1
CLR
READY.
```

Note that the CLOSE 1 didn't get the READY. back. It took the CLR to return BASIC's messages to the screen. If you enter LIST, the program will appear on the screen. Rewind the tape and RUN. Three asterisks now appear after the RUN. These were printed by the program. This is one reason I don't trust my PET after a CMD. The text between the OK and the ending READY was found as a data file.

When the PET was under the influence of CMD, the letters you typed in were put onto the screen. This echoing is done by the PET's operating system, so CMD won't put these out to the device.

Though CMD looks like a good way to LIST program to tapes as data files, there is a snag. My example is shorter than 191 characters, and a LIST via CMD isn't smart enough to "jiffy" the data tape (this has been fixed on the new PETs). You run the risk of losing tape records when you try to read an "unjiffied" tape.

Try to verify that CMD 1,"HELLO OUT THERE" will print HELLO OUT THERE onto the tape. Remember that if you CMD a device on the IEEE 488 bus, any PRINT# to the bus will require a repetition of the CMD if you want the device to remain in the listening state.

Talking to the Clock Again

(For a description of the HP clock see part 1 of this article.)

First, you must check the device address on the DIP switch (which will be near the 488 female connector) and make sure the address is in the range 4 to 15. The enter a short program (Example 2) into the PET. This program consists of three subroutines to facilitate communicating with the clock. Remember that the PET will not accept an INPUT statement as a direct command.

```
 10  OPEN 1,7
 20  RETURN
100  INPUT"SAY TO CLOCK:";S$
110  PRINT#1,S$
120  RETURN
200  INPUT#1,C$
210  PRINT"CLOCK SAYS: ";C$
220  RETURN
```

Example 2.

First, enter GOSUB 10 as a direct command. This opens file 1 to device 7, which is our clock on the IEEE bus. OPEN merely sets things up; nothing is sent to the bus yet.

To read the time, enter GOSUB 200:

```
GOSUB 200
CLOCK SAYS: 0103020204 (Jan. 3, 2:02:04 AM)
```

Your PET might give ? SYNTAX ERROR after this operation. This is a harmless feature of the PET.

To set the clock, using Jan. 29, 9:17 PM, as our example, enter:

```
GOSUB 100
SAY TO CLOCK? RDDDDDDDDDDDDDDD
DDDDDDDDDDDDD(28 Ds)
```

The clock starts at day 1. To set to day n, use n – 1 Ds. To set the hour, enter the following.

```
GOSUB 100
SAY TO CLOCK? HHHHHHHHHHHHHHH
HHHHHH(21Hs)
```

Minutes and seconds are set similarly.

```
GOSUB 100
SAY TO CLOCK? MMMMMMMMMMMM
MMMMSSS (17Ms, 3Ss)
```

We are now set to 9:17:03. When I did this by hand, the clock moved forward about a minute, so the number of M's used should be changed to accommodate for this.

Talking to the HP 8165A Programmable Signal Source

(For a description of the HP 8165A, see part 1 of this article.)

The 8165A is a fine instrument with many switches, knobs, buttons and options and a correspondingly wide array of IEEE 488 commands (see Fig. 12, part 1).

The precise contents of each example concern the 8165A, which is an instrument you will probably never meet! My intention is to show you how direct mode commands—that is, BASIC statements without line numbers—can be used to control an instrument and help in debugging.

First, I hooked the 8165 to the 488 cable, and the PET turned on. The 8165 was addressed to 8. When the PET came on, IFC was true for about one second. This put the 8165 in local mode, where the front panel works as usual. Many instruments will ignore their front panels when the 488 bus addresses them. Once the PET addresses the 8165, you cannot control it from the front panel anymore. (An LED indicates this on the 8165.)

The following short program takes care of input from the instrument:

```
10  INPUT#1, A$
20  PRINT A$
```

This substitutes for the illegal direct command (INPUT#1,A$: PRINTA$), which I would like to use, but the PET forbids (try it and see!).

Since I wanted the 8165 to output a 1 kHz sine wave at an amplitude of 1.5 volts, I used the following IEEE commands:

F1—Set to sine wave
FRQ 1 kHz—Set frequency
AMP 1.5 V—Set amplitude
I1—Set to normal operation (continuous signal output)

First, open the IEEE file:

```
OPEN 1,8
READY.
```

Then send the settings:

```
PRINT #1,"F1"  (At this point, the "Remote" LED went on, and I can no longer work the front panel.)
PRINT #1,"FRQ1KHZ"
PRINT #1,"AMP1.5V"
PRINT #1,"I1"
```

Nothing happened! My scope showed only a flat trace! Upon reviewing my steps, I noticed that I overlooked the Disable Output (OD) and Enable Output (OE) commands. I entered PRINT #1,"OE", and a sine wave appeared on the scope.

You could also send this setting as one string. For example, PRINT #1,"F2FRQ1.2KHZAMP 1.2VI1OE" sets up a 1.2 kHz triangle wave at 1.2 V amplitude.

The 8165 can also report some of its switch settings. Now we can use the tiny program in the PET:

```
GOTO 10
F1 D2 12 FM0 AM0
```

Since the PET has difficulty with GOSUB in direct mode and the IEEE bus, we must make a program change:

```
10  INPUT#1, A$
20  PRINT A$
30  RETURN
```

We will quickly be reminded that any time we change a program, all the variables, including opened files, will be lost:

```
GOSUB 10
?FILE NOT OPEN ERROR IN 10
```

So we try again:

```
OPEN 1,8
GOSUB 10
F1 D2 I2 FM0 AM0
?SYNTAX ERROR IN 22066
```

The PET will provide the ?SYNTAX ERROR about 90 percent of the time when the IEEE is accessed via the INPUT# statement and the PET is executing a directly called subroutine. However, this doesn't appear to affect anything. I avoided this by not making the little program a subroutine the first time.

So, if you are in a pinch, remember that the PET's direct command capability can rescue you with IEEE 488 devices and provides an inexpensive way to explore a new instrument.

Talking to More than One Device

Now that each of the instruments has been in the bus individually, the next step is to try the 488 with both of them on at the same time. I connected the HP clock and the 8165 to the 488 bus and gave the clock address #7, and the 8165 address #8. Then I entered the short program for INPUTs:

```
10  INPUT #1, A$
20  PRINT A$
30  END
100  INPUT #2, B$
110  PRINT B$
120  END
```

First, OPEN the files:

```
OPEN 1,7
OPEN 2,8
```

If you get a ?FILE OPEN ERROR, just enter CLR and start over.

Taking a peek at the clock resulted in:

```
GOTO 10
0130051957      (30 Jan., 5:19:57)
```

And peeking at the 8165 gets me:

```
GOTO 100
F1 D2 I2 FM0 AM0
```

which is the usual mystery message that the 8165 says to me. There isn't any point in explaining this message, for your instrument will say something different and meaningful only to you.

PRINT #1 and PRINT #2 will

work just fine, and so two instruments and the PET can live in harmony together.

A Gotcha

I decided to turn off the 8165 with the PET set up for two instruments as described above. Sure enough, strange things happened.

The clock worked fine:

```
GOTO 10
0130052525
```

And just for fun, look what happens with the 8165 (which isn't on):

```
GOTO 100
F1 D2 I2 FM0 AM0
```

The 8165 has some internal batteries to store and memorize settings until it is turned on again. It also will respond to the IEEE 488 bus.

Now to try things in reverse—the clock doesn't have any batteries. (Clock is off; 8165 is on.)

```
GOTO 100
F1 D2 I2 FM0 AM0        8165 is fine
GOTO 10
F1 D2 I2 FM0 AM0        What's this?
```

The 8165 will reply to any address if it is the only device on the bus. The clock acts in the same way. (I don't know if this is a PET fault or an HP design decision. Check your device.)

If your program is intended for more than one device, this can be a disaster. Make sure all required devices are operating when using multiple devices on the bus.

I ran into another gotcha: the 8165 wouldn't accept every frequency change. I tracked this problem down to the presence of the HP clock on the bus. When I turned the clock off, everything worked fine. When debugging, remember to have only one device on your bus.

Common Errors

In theory, if you have understood everything to this point, you can now get an IEEE 488 instrument and make it play with your PET. In practice, this won't happen.

Finding errors is the hardest part of programming, and when you work with the IEEE bus, you can make many mistakes that don't look like errors. When you are able to see errors easily and immediately, you won't need this article.

Here is an incomplete list of the common errors in wait for the unwary IEEE/PET programmer.

The misplaced address. The PET's IEEE addresses are from 4 through 30. The addresses 0 to 3 are reserved for the PET's other I/O devices:

0—Keyboard
1—Tape unit #1
2—Tape unit #2
3—Video screen

If you OPEN a file to the reserved addresses, you won't be speaking to the IEEE bus!

If a device isn't running when the PET wants to talk to it, you will usually get a ?DEVICE NOT PRESENT ERROR. However, if some other device is operating on the bus, you might get the other device's response instead. This happened to me with the HP clock and the 8165. If one was turned off, the other would respond, even though the OPEN statement was referring to the inactive device. This can badly confuse your program.

Time-outs. The PET will only wait for 64 milliseconds before giving up on a device that is slow to respond to the IEEE 488 handshake. Though the IEEE 488 is supposed to work at *any* speed, you may wonder what to do if a device on the bus has failed. If the PET were to wait for a response, there would be no way to return to the user. The 64 ms interval was chosen from the timers available on the 6522 VIA chip, which can count up to 65535 at the 1 MHz clock rate of the PET.

Most instruments will respond within the 64 ms interval, and the PET will read and write the data correctly. This was true of the HP instruments at my disposal. To exercise the PET time-outs, I attached both the clock and the 8165 to the bus, and then OPENed a file to a nonexistent address:

```
NEW

10  INPUT#3,A$
20  IF ST THEN PRINT"ST IS" ST
30  PRINT A$
40  A$ = ""

OPEN 1,7      (Open the clock to file 1)

OPEN 2,8      (Open the 8165 to file 2)

OPEN 3,10     (The nonexistent device)
```

The little program attempts to input from the nonexisting device. The ST value is a reserved BASIC variable used by the PET for indicating I/O conditions. If ST isn't zero, something went awry.

Now to talk a bit to the devices to wake them up:

```
PRINT #1,"R"      (And the clock resets)
PRINT #2,"E0"     (And the 8165 puts out
                   a signal)
```

If a look at ST is made, all's well:

```
PRINT ST
 0
```

This may take a few tries to work right.

Now to try that nonexistent device:

```
PRINT #3, "HELLO"
```

Looks OK, right? Well, let's see . . .

```
PRINT ST
 -128
```

This is the PET's ST code for "device not present."

Now to try the little program:

```
GOTO 10
ST IS 2

READY.
```

The ST code is 2, which is the time-out for reading data; the nonexistent device didn't say anything. Recall that line 30 said to print A$. The PET *did* print A$, which was an empty string.

The solution to this dilemma is to keep on trying! Write a loop that redoes the INPUT# or PRINT#. In most cases, a slow device will send its characters rapidly enough—once it has its message ready.

Consider these two sample loops:

```
100  PRINT #5," some message or other"
110  IF ST = 1 THEN 100
200  INPUT #6,B$
210  IF ST = 2 THEN 200
```

If you want to mask for certain bits, you can use the AND operator, but parentheses are needed. The above examples would read:

```
110  IF (ST) AND 1 THEN 100     and
210  IF (ST) AND 2 THEN 200
```

The removal of the parentheses makes the PET see the expression as:

```
IF ST AND 1    looks like    IF S TAN D 1
```

which will result in a ?SYNTAX ERROR. Use parentheses or rearrange the order of operations in these cases.

The literal principle. PET outputs to a file the same characters that it sends to the screen. This is also true for the IEEE 488. The PET's format for PRINTing a number is:

(space or – sign) (digits) (optional exponent) (cursor right)

This can raise havoc with an IEEE device that is expecting a character after the number.

Consider the following example:

```
10  PRINT "clr";      (clear screen)
20  FOR J = 1 TO 10
30  PRINT "**********"
40  NEXT J
50  PRINT "hm";       (home cursor)
60  FOR J = 1 TO 10
70  PRINT J"IS A NUMBER"
80  NEXT J

RUN

 1 IS A NUMBER*****
 2 IS A NUMBER*****
 3 IS A NUMBER*****

  etc
```

The asterisk after the number comes from the cursor right character that was sent to the screen. The cursor right follows any numbers sent to the IEEE 488 bus.

The following program sets the frequency of the 8165.

```
10  OPEN 1,8   (The 8165 is at address 8)
20  FOR J = 1000 TO 2000 STEP 10
30  PRINT #1,"FRQ"J"HZ"
40  FOR K = 1 TO 1000
50  NEXT K     (This is a 3 second delay
               loop)
60  NEXT J
```

When this is RUN, the 8165 gives all signs of distress. The frequency appears on the front panel, but the LED that indicates correct entry stays blinking (not completed). Also, the scope shows no change. The PET screen blinks at intervals, indicating that EOI is made true now and then. (I suspect the instrument is making this happen.)

The following modification will fix this:

```
30  PRINT#1,"FRQ"STR$(J)"HZ"
```

The STR$ function converts a number to the string that would be PRINTed, without the cursor right at the end! The general fix for numbers is simple: convert *all* numbers to strings before putting on the IEEE 488 bus.

Fractions. Now that the frequency example is working right, how about trying some other STEP sizes. Here is a simple change:

```
20  FOR J = 1 TO 2 STEP .01
30  PRINT #1,"FRQ"STR$(J)"KHZ"
```

The J loop was changed to do the same thing, but in kilohertz. Line 30 was changed to reflect this. When RUN, it all works fine until about 1.25 kHz—the 8165 now shows 1.259 kHz instead of 1.260. A look at J gives us the clue we need:

```
BREAK IN 40      (Press STOP key)
PRINT J
 1.25999999
```

The PET slips up when computing with fractions . . . and this eventually shows up. The fraction .01 becomes a repeating binary decimal, and after repeated addition, the round-off appears as a slight reduction of the number being added to. In this case, 1.260 turns into 1.25999999.

Catching this is easy . . . if J were put onto the screen first!

```
35  PRINT STR$(J)
```

If you do this, the first "blow up" comes at 1.22999999. Now you are faced with a programming problem: how to get around nasty numbers. One way is to take the INT function, such as:

```
STR$(INT(J*100 + .5)/100)
```

which rounds the number in the hundredths place. More complex tricks will be needed if the PET insists on scientific notation, such as

```
2.35E - 03
```

PRINT your IEEE output onto the screen while debugging.

Next month, we will wrap up our three-part series with a further look at the programming style with the IEEE 488. ■

The PET IEEE 488 File I/O Statements

The PET sees the IEEE 488 bus as a file, and the file I/O statements apply to IEEE 488 transfers. Be sure you know the cassette file I/O before tackling the IEEE 488 bus.

The PET file I/O statements are:

● OPEN (file number), (device number), (secondary address), (filename)

OPEN instructs the PET to associate the file number with the desired I/O device. BASIC uses the file number in its PRINT#, INPUT# and GET# statements to determine where the I/O is to take place. The file number may be from 1 to 255.

The device numbers are assigned as follows:

0 – Keyboard
1 – Cassette unit #1
2 – Cassette unit #2
3 – Screen
4 – 30 IEEE 488 bus

This implies that your IEEE device must be addressed in the range of 4 to 30. Most IEEE devices have a switch or jumpers that permit the changing of their addresses.

The secondary address and filename are optional. However, if you want to use the filename, the secondary address must also be included. The secondary address has the range of 0 to 31.

If the filename is not specified, the OPEN statement sends nothing to the IEEE 488 bus. When BASIC sees the PRINT#, INPUT# and GET# statements, the device number (and secondary address, if specified) are put on the IEEE bus as part of the usual transfer sequences.

If a filename is specified, (i.e., A$ or "SOME NAME"), the OPEN statement activates the IEEE bus making ATN true and sends:

LISTEN (to the appropriate device)
SECONDARY ADDRESS (ORed with 11110000)
FILENAME (all characters)

This permits suitably complex command sequences that require ATN to be true to be sent. If the command sequence has to be repeated later, CLOSE the file and OPEN it again. I haven't been able to check if the above assertions about the filename are true. If you have a bus analyzer, check this out!

● PRINT# (file number), (values to be sent)

First, don't use the abbreviation ?# ; it won't work (when executed, you will see ?SYNTAX ERROR) and will list as PRINT#. Spell out PRINT completely!

The PRINT# sets ATN true and sends the device number as a LISTEN address. If a secondary address as specified, it will be sent also. The device number and secondary address are taken from the appropriate OPEN statement.

ATN is then made false, and the values to be sent are transmitted as ASCII characters in exactly the same way as they would be sent to the screen. For example, if a number is sent, a cursor right character follows the last digit. If you use "," to separate columns, lots of cursor rights are sent. If the PET feels a number should be in scientific format (i.e., 1.53E – 07), that's what is sent! EOI is made true with the last character of data sent.

After the values are sent, an UNLISTEN is sent (with ATN true), and all listening devices are set free.

● INPUT# (file number), (values to be input)

INPUT# sets ATN true and sends the device number as a TALK address. If a secondary address was specified, it will be sent too. The pertinent OPEN statement is used for these values.

ATN is then made false, and the PET accepts characters from the device to the PET's input buffer. If the talker activates EOI, a carriage return is added to the end of the buffer.

After the characters are accepted and carriage return or EOI is recognized, the PET sets ATN true and sends an UNTALK, which releases the device.

BASIC then scans the input buffer in the same way that an ordinary INPUT statement looks at what is typed in. This means that commas and quotes will have the same effects as with normal INPUT. It is best to use an INPUT (string) form and hope your device doesn't send any commas!

As with cassette INPUT#, an 80-character buffer is used. If more than 79 characters arrive without a carriage return, the PET will go into "limbo," and all is lost. (New PETs have this fixed. Over 80 characters are ignored (or worse, the buffer is initialized, and the first 80 characters are lost!). If you have a new PET, try it with cassettes and find out what happens.

INPUT# is susceptible to "time out," and ST should be checked for a time out. Repeat the INPUT# if a time out is detected.

● GET# (file number), (value for entry)

GET# sets ATN true and sends the device number as a TALK address and the secondary address, if specified. ATN is made false, and a single character is accepted.

Then, the UNTALK with ATN true is sent, and the character given to BASIC. For the reasons that make GET X unusable, be sure to only use the GET# (string) form.

The assertion of the UNTALK after GET# makes transmission of multicharacter messages from devices impractical, as most devices will try to repeat their message on repeated application of GET#.

As with INPUT# ST should be checked for a time out, and if timed out, the GET# should be repeated.

● CLOSE (file number)

CLOSE releases the I/O assignments. The PET will allow a maximum of ten files OPEN at one time, and CLOSE will let you reuse an I/O assignment. If you OPEN more than ten files, old PETs will go into limbo and all will be lost. New PETs presumably have this fixed.

If the corresponding OPEN statement had a filename specified, CLOSE sets ATN true and sends the device number and secondary address (ORed 11100000). This feature is intended for PET peripherals.

● CMD (file number), (values to be sent)

CMD initiates the same sequence as PRINT# and sends the values, if any, in the same way that PRINT# does. When finished, CMD does not send the UNLISTEN, so any devices addresses with CMD will listen to further CMDs or PRINT# to the IEEE bus.

All of BASIC's output will be routed to the device defined in the OPEN statement for the file number. If the PET is in command mode, this includes the READY., error messages and LIST. If in run mode, any BASIC printouts, from PRINT to the screen, will go to the IEEE bus instead. A PRINT# will recover from the effects of CMD.

If you are using CMD in command mode, the cursor may not echo the RETURNs you press. The PET will "echo" your keystrokes, but any outputs from BASIC will vanish to the IEEE device. The PRINT# to your IEEE device is the safest recovery from CMD. Remember that any editing of a BASIC program will destroy all variables. This includes open files and CMDs.

● ST (status word)

After each I/O operation, the PET sets the value of a special variable named ST, which will hold its value until the next I/O operation. So the best policy is to check it immediately! The values of ST for the IEEE bus are:

1 Timeout on write
2 Timeout on read (This one should *always* be checked)
64 EOI true
– 128 Device not present

The PET waits for 64 milliseconds to see if a device will respond to the IEEE handshake. If the device doesn't, the I/O operation is quietly aborted, and ST is set. If you are INPUT#ing, you will get "nothing" or zeroes back. If you are PRINT#ing, everything seems to be all right. If your device is slow to respond, checking ST is mandatory.

PRINT#, INPUT# and GET# will return the ?DEVICE NOT PRESENT error if the bus is in an illegal state (which is true if the bus has no devices or the LISTEN or TALK isn't responded to). ST will also be set.

● LOAD, SAVE and VERIFY

The old PETs have a severe error in their IEEE software which prevents the functioning of LOAD, SAVE or VERIFY. The ATN line was left true during the data part of the transfer. This is why owners of old PETs who purchase the PET disk get the new ROMS; the disk won't function with the old ROMs.

The format is the same as with tapes:

LOAD (filename), (device number)
SAVE " " "
VERIFY " " "

Once the IEEE bus is set to listen or talk, the first four bytes must contain the beginning and ending address + 1 of the block to be transferred. The transfer is then done as pure binary until finished. The bus is then released with an UNT or UNL as needed.

VERIFY will say ?VERIFY ERROR and set ST to 16 if any mismatches were found between the incoming data and the core image in the PET's memory. Since my PET is an old model with the original ROMs, I haven't been able to check LOAD, SAVE and VERIFY for the IEEE 488 bus.

Get Your PET On the IEEE 488 Bus

The final stop on this three-part tour.

Gregory Yob
Box 354
Palo Alto, CA 94302

Commodore's printer and disk use the secondary addresses to control special functions within each device. The secondary address extends the range of allowable addresses on the IEEE 488 bus and is included after the LISTEN or TALK address with ATN made true. Most IEEE devices do not use secondary addresses.

The secondary address permits the device to distinguish between data transfers (for example, file I/O via the disk) and command sequences (for example, to initialize a new disk). The following is a brief summary of the secondary addresses used by Commodore's devices.

PET Printer.

0 – Normal printing. The printer accepts characters and prints them as received.

1 – Formatted printing. The characters are accepted and rearranged according to an internally stored format specification.

2 – Format specification. The characters specifying the format to be used are accepted by the printer.

3 – Pagination control. Accepts a number indicating the number of lines per page.

4 – Control of diagnostic messages. If desired, diagnostic messages will be printed when errors are found. For example, if a number overflows its format, a message indicating this will be printed. This secondary address controls the options to use this feature.

5 – Load programmable character. The printer accepts bytes that specify the dot matrix for one programmable character.

PET Disk.

2 to 14 – Disk "channels" data transfers. The PET disk can have from zero to five files open at once. Each file is defined with an OPEN statement of the form:

OPEN (Log Addr), (Device Addr), (Channel Number), (Command String)

The channel number is a secondary address in the range of 2 to 14. The command string specifies the file type and drive. For example, "0,FILEONE, SEQ, WRITE" means open the file named FILEONE on drive 0 as a sequential file for write only access.

15 – Disk command channel. A variety of commands to the disk is sent via PRINT# to a file opened to the secondary address of 15. The disk can also send error and diagnostic messages to the PET through this channel.

Though it is possible to control complex devices in this manner, these methods can become awkward and clumsy if many data transfers are needed, as is the case for disks and printers. Commodore chose this method to avoid having to modify or extend the PET's BASIC.

Ironically, Commodore now offers a machine-language program, WEDGE, which functions as an extension to BASIC for control of the PET Disk.

Two Examples

In most applications of IEEE instruments, your task will extend beyond communicating with the device. Once communications with the device are established, there remains the conversion of the data to a form usable by people or some other instrument that uses a different form of data. Also, care should be taken to make human communications as pleasant as possible. If your application is in a production (that is, for daily use, and not as an occasional experiment), clarity and reliability are important.

Two BASIC programs, which illustrate how the HP Clock and the HP Signal Source might be used in real-life situations, follow. They are presented here as examples of programming style with the IEEE 488.

Example 1: The HP Clock

Part 1 (*Microcomputing*, July 1980) describes the codes used for the HP Clock with the IEEE 488 bus. Listing 1 interacts with the HP clock in a "human-workable" form. Let's first take a look at how the program is seen from the outside (often called "human engineering" or "the user interface").

When the program is RUN, the following message appears on the screen:

```
HP CLOCK PROGRAM
PRESS ANY KEY WHEN YOU HAVE THE
CLOCK CONNECTED VIA THE IEEE 488
AND THE POWER ON.
```

This reminds the user to connect the clock on the bus and turn on the clock's power. If the PET tries to address a device that isn't connected or turned on, the ?DEVICE NOT PRESENT error message will appear and stop the program. Unfortunately, there is no graceful way to prevent this and keep the program running (some versions of BASIC have error traps; i.e., ON ERROR 5 GOTO . . .).

After you press a key, the request appears:

Listing 1. HP Clock program.

```
10 REM NICE HP CLOCK PROGRAM
20 PRINT"clr HP CLOCK PROGRAM"
30 PRINT"dn dn PRESS ANY KEY WHEN YOU HAVE THE
40 PRINT"CLOCK CONNECTED VIA THE IEEE 488
50 PRINT"AND THE POWER ON.
60 GET A$:IFA$="" THEN 60
70 REM INITIALIZE
80 DIM M$(12),M(12)
90 FOR J=1 TO 12:READ M$(J),M(J):NEXT
100 DATA JAN,31,FEB,28,MAR,31
110 DATA APR,30,MAY,31,JUN,30
120 DATA JUL,31,AUG,31,SEP,30
130 DATA OCT,31,NOV,20,DEC,31
140 INPUT"dn dn CLOCK'S DEVICE ADDRESS:",AD
150 IF AD<3 AND AD>16 THEN 170
160 PRINT"SORRY, LEGAL ADDRESSES ARE 4 - 15":GOTO 140
170 OPEN 1,AD
180 INPUT"dn dn IS THIS A LEAPYEAR";L$
190 IF LEFT$(L$,1)="Y" THEN M(2)=29:PRINT"BE SURE TO SET
    THE CLOCK TO 366 DAYS"
200 REM TIME SETTING REQUEST
210 INPUT"dn dn SET THE TIME";L$
220 IF LEFT$(L$,1)="Y" THEN GOSUB 1000
230 REM DISPLAY TIME
240 GOSUB 2000
250 GOTO 210

1000 REM TIME SETTING ROUTINE
1010 PRINT"clr sp SET THE DATE"
1020 PRINT"dn dn ENTER MONTH AND DAY IN THE FORM:
1030 PRINT"dn sp sp sp sp sp MONTH (SPACE) DAY
1040 PRINT"dn FOR EXAMPLE: sp sp MARCH 25
1050 INPUT"dn";MD$
1100 REM PARSE OUT MONTH & DAY
1110 M1$=LEFT$(MD$,3)
1120 FOR MN=1 TO 12
1130 IF M1$=M$(MN) THEN 1200
1140 NEXT MN: PRINT"dn dn I DON'T RECOGNIZE THE MONTH.
1150 PRINT"PLEASE SPELL THE MONTH COMPLETELY.
1160 PRINT"dn dn PRESS ANY KEY TO TRY AGAIN
1170 GETA$:IFA$="" THEN 1170
1180 GOTO 1010
1200 FOR J=1 TO LEN(MD$)
1210 IF MID$(MD$,J,1)=" sp " THEN 1300
1220 NEXT J
1230 PRINT"dn dn YOU FORGOT THE DAY
1240 GOTO 1160
1300 DY=VAL(MID$(MD$,J))
1310 IF DY>Ø AND DY<M(MN)+1 THEN 1400
1320 PRINT"dn dn YOUR DAY IS INCORRECT. IT MUST BE
1330 PRINT"FROM 1 TO"M(MN)"."
1340 GOTO 1160
1400 REM COMPUTE NUMBER OF DAY TICKS
1410 DT=Ø: IF MN=1 THEN 1430
1420 FOR J=1 TO MN-1: DT=DT+M(J):NEXT J
1430 DT=DT+DY-1
1450 REM DT IS # OF DAYS TO ADVANCE

1500 PRINT"clr sp SET THE TIME"
1505 PRINT"dn dn ENTER THE TIME IN THE FORM:
1510 PRINT"dn sp HOUR : MINUTE : SECOND : AM OR PM
1520 PRINT"dn FOR EXAMPLE: sp sp 2:25:36:PM"
1530 PRINT"dn";:GOSUB 4000
1600 REM PARSE OUT HOURS,MINS,SECS,AM?PM
1610 T$=T$+"XX":TH=VAL(T$)
1620 GOSUB 3000: IF PT>Ø THEN 1700
1630 PRINT"dn YOU DIDN'T INCLUDE EVERYTHING
1640 PRINT"PLEASE ENTER ALL FOUR ITEMS WITH
1650 PRINT"COLONS BETWEEN EACH OF THEM
1660 PRINT"dn PRESS ANY KEY TO TRY AGAIN
1670 GETA$:IFA$="" THEN 1670
1680 GOTO 1500
1700 T$=MID$(T$,PT+1)
1710 TM=VAL(T$)
1720 GOSUB 3000: IF PT=Ø THEN 1630
1730 T$=MID$(T$,PT+1)
1740 TS=VAL(T$)
1750 GOSUB 3000: IF PT = Ø THEN 1630
1760 T$=MID$(T$,PT+1,2)
1800 REM ERROR MESSAGES
1810 IF TH<1 OR TH>12 THEN PRINT"dn dn YOUR HOURS MUST
     BE FROM 1 TO 12":GOTO 1660
1820 IF TM<Ø OR TM>59 THEN PRINT"dn dn YOUR MINUTES
     MUST BE FROM Ø TO 59":GOTO 1660
1830 IF TS<Ø OR TS>59 THEN PRINT"dn dn YOUR SECONDS
     MUST BE FROM Ø TO 59":GOTO 1660
1840 IF T$="AM" OR T$="PM" THEN 1860
1850 PRINT"dn dn PLEASE USE AM OR PM ONLY":GOTO 1660
1860 REM AM/PM LOGIC
1870 IF T$="AM" AND TH=12 THEN TH=Ø
1880 IF T$="PM" AND TH<12 THEN TH=TH+12
1900 REM SET CLOCK AT LAST
1910 PRINT#1,"RP";
1920 IF DT>Ø THEN FOR J=1 TO DT:PRINT#1,"D";:NEXT
1930 IF TH>Ø THEN FOR J=1 TO TH:PRINT#1,"H";:NEXT
1940 IF TM>Ø THEN FOR J=1 TO TM:PRINT#1,"M";:NEXT
1950 IF TS>Ø THEN FOR J=1 TO TS:PRINT#1,"S";:NEXT
1960 PRINT#1,"T"
1970 RETURN

2000 REM DISPLAY TIME
2010 PRINT"clr sp sp sp sp sp THE CURRENT TIME IS
2020 PRINT"dn dn sp sp DATE:"
2030 PRINT"dn dn sp sp TIME:"
2040 PRINT"dn dn dn dn dn dn dn dn dn dn dn dn dn dn      (14 dn's)
       sp sp PRESS ANY KEY TO SET TIME
2050 GETA$:IFA$<>"" THEN RETURN
2060 REM FETCH TIME
2070 INPUT #1,T$
2080 IF LEFT$(T$,1)="?" THEN GOSUB 5000
2090 REM PARSE OUT PARTS
2100 T1=VAL(MID$(T$,1,2))
2110 T2=VAL(MID$(T$,3,2))
2120 T3=VAL(MID$(T$,5,2))
2130 T3$=MID$(T$,5,2)
2140 T4$=MID$(T$,7,2)
2150 T5$=MID$(T$,9,2)
2160 PRINT"hm dn dn dn rt rt rt rt rt rt rt rt rt"M$(T1);T2
2170 REM AM/PM CALCS.
2180 T6$="AM"
2190 IF T3>11 THEN T6$="PM"
2200 IF T3>12 THEN T3=T3-12
2210 IF T3=Ø THEN T3=12
2220 T3$=RIGHT$(STR$(T3),2)
2250 PRINT"dn dn rt rt rt rt rt rt rt rt"
     T3$":"T4$":"T5$":"T6$
2500 GOTO 2050

3000 REM SCAN T$ FOR COLONS
3010 FOR PT=1 TO LEN(T$)
3020 IF MID$(T$,PT,1)=":" THEN RETURN
3030 NEXT PT
3040 PT=Ø: RETURN
4000 REM FETCH STRING VIA GET DUE TO
4010 REM FLAKEY PET INPUT STATEMENT
4015 T$=""
4020 GET A$: IF A$<>"" THEN 4100
4030 PRINT"rvs sp lft";:FOR J=1 TO 300: NEXT
4040 GET A$: IF A$<>"" THEN 4100
4050 PRINT"off sp lft";:FOR J=1 TO 300: NEXT
4060 GOTO 4020
4100 PRINT"off sp lft";
4110 IF A$=CHR$(13) THEN PRINT: RETURN
4120 PRINT A$;: T$=T$+A$: GOTO 4020

5000 PRINT"hm dn dn dn dn dn dn dn dn dn dn dn dn
     >>>>>>> sp TIME NEEDS TO BE SET sp<<<<<<<"
5010 PRINT"dn>>>>>>> sp DUE TO POWER FAILURE sp <<<<<<<"
5020 T$=MID$(T$,3):RETURN
```

CLOCK'S DEVICE ADDRESS;?

Now enter the address on the DIP switches for the device. If an unacceptable value, such as 16, is entered, the PET will respond with:

SORRY, LEGAL ADDRESSES ARE 4-15

and ask again. The best way to avoid problems is to forbid illegal values for inputs, tell the user that he has goofed and mention the correct range of values.

Once the device address is in, the PET asks:

IS THIS A LEAP YEAR?

If "YES" is entered, a reminder appears to set the clock accordingly.

BE SURE TO SET THE CLOCK TO 366 DAYS

The last request asks:

SET THE TIME?

If the user doesn't want to set the time, the screen clears and the date and time are shown:

THE CURRENT TIME IS
DATE: JAN 29
TIME: 7:02:54 PM

PRESS ANY KEY TO SET TIME

The time ticks away with the seconds changing the most rapidly. A different set of values will appear on the clock:

01 29 19 02 54

The program has translated from 24-hour time to normal AM/PM time and changed the month from a number to the month's name.

The HP clock will send a ? as the first time character if the clock has not been set since a loss of power. If you pull the plug on the clock and plug it in again, the program will stop with a ?DEVICE NOT PRESENT ERROR. When the program is RUN, the time will be displayed with the following in the space

between the time and the PRESS ANY KEY line:

```
>>>>>TIME NEEDS TO BE SET<<<<<
>>>>>DUE TO POWER FAILURE<<<<<
```

Now if you press a key, the SET THE TIME? request will reappear:

```
SET THE TIME? YES
```

The screen clears and will display:

```
SET THE DATE
ENTER MONTH AND DAY IN THE FORM:
     MONTH (SPACE) DAY
FOR EXAMPLE: MARCH 25
? JANUARY 29
```

If the first three letters in the month are incorrect, the program will make you start over:

```
I DON'T RECOGNIZE THE MONTH.
PLEASE SPELL THE MONTH COMPLETELY.
PRESS ANY KEY TO TRY AGAIN.
```

If you missed the date, the PET says:

```
YOU FORGOT THE DAY
PRESS ANY KEY TO TRY AGAIN
```

If you enter an inappropriate date, such as JAN 45, the PET, will say:

```
YOUR DAY IS INCORRECT. IT MUST BE FROM 1 TO 31.
```

The program has the number of days for each month stored inside. If the month were February, the range 1 to 28 would have been shown instead.

Now that the date is entered correctly, the screen clears to let the time be entered.

```
SET THE TIME
ENTER TIME IN THE FORM:
HOUR : MINUTE : SECOND : AM OR PM
FOR EXAMPLE: 2:25:36:PM
7:19:25:PM (you enter this line)
```

The screen will flicker a bit, and then the time display will appear.

The PET won't correctly input a string with colons in it, so the entry here is "faked" to look like a normal INPUT line. Unfortunately, if you must INST or DEL to correct your line, the correction won't really be entered. This can be programmed around, but I didn't feel like doing it with an instrument on loan to me for a week. The subject of faking INPUT is an article in itself.

Again, there are some error messages to help and assist the user:

```
YOU DIDN'T INCLUDE EVERYTHING
PLEASE ENTER ALL FOUR ITEMS WITH
COLONS BETWEEN EACH OF THEM
PRESS ANY KEY TO TRY AGAIN
YOUR HOURS MUST BE FROM 1 TO 12
YOUR MINUTES MUST BE FROM 0 TO 59
YOUR SECONDS MUST BE FROM 0 TO 59
PLEASE USE AM OR PM ONLY
```

Here, a bad entry only forces you to reenter the time. The date is OK, so why redo it?

Perhaps this example is extreme. In many situations it isn't worth the programming time to make a program completely convenient to use. As an idealist, I wrote it up to show what can be done if ease of use is required.

HP Clock BASIC Program Review (Listing 1)

Lines 10 to 60 announce the program and force the user to wait until he has made sure the HP Clock is attached to the PET's IEEE 488 and the power is turned on. DATA in lines 100 to 130 are placed in the months' names' array M$ and the months' lengths' array M.

Lines 140 to 170 request the HP Clock's address and check to see if the address is legal. Line 160 tells the user to try again and mentions the legal range as a hint. Lines 180 and 190 take care of the leap-year problem by changing the month length for February to 29 days and reminds the user to check the leap-year switch on the HP Clock.

In lines 200–220, the user is asked if the time is to be set (which must be done when the clock is first used), and a loop is entered in lines 240 and 250. Subroutine 1000 sets the time, and subroutine 2000 displays the time. The program will not leave subroutine 2000 until a key is pressed. Line 250 jumps to the time-change request as needed.

Setting the time in subroutine 1000 is a complicated job, requiring correctly entering the data. First, you must enter the month and day as explained in lines 1010 to 1040, which give an example of the expected format.

Line 1050 picks up the user's entry, and lines 1000 to 1180 take a look at the first three characters to see if they fit a month's name. Lines 1140 to 1180 take care of any mistake in the entry of a month's name.

Lines 1200 to 1220 scan the input string, MD$, until a space is found. This removes the remnants of the month's name and brings us up to the date digits. Failure to find a space means the day was forgotten, and the user is told to start all over.

Lines 1300 to 1340 check the day number with the number of days in the month M(MN). If everything is OK, lines 1400 to 1450 will figure out the value DT, which is used to send the correct number of Ds. to the clock for date setting.

Now that we have the number of days from Jan. 1 (in the number DT), lines 1500 to 1530 will tell the user to enter the time in a familiar format—HH:MM:SS:AM or PM. Subroutine 4000 is used to enter the string T$ via the GET statement. In lines 1620 to 1850, the string T$ is snipped apart at the colons, and each part is examined for the correct range of values; subroutine 3000 looks for the colons, and lines 1680 to 1760 do the scissor-work. We eventually end up with the values TH, TM, TS and T$, for hours, minutes, seconds and AM/PM values.

Lines 1860 to 1880 adjust the hours, TH, according to the AM or PM value. Lines 1900 to 1970 set the HP Clock—first the clock is reset via "RP," and then the correct numbers of "D," "H," "M" and "S" are sent to set the time. Then "T" is sent to start the clock.

Subroutine 2000 sets up the screen in lines 2010 to 2060. Note that the GET in line 2050 only checks if a character was entered. If not, it will continue to line 2070. The HP Clock is accessed in line 2070, and line 2080 checks for "?." The "?" means the clock saw a power failure, and subroutine 5000 will warn of this event.

Lines 2100 to 2150 get the various parts of the HP Clock's message. T1 is the month number; T2 is the day number. Line 2160 displays the month and day values.

Lines 2170 to 2220 adjust the hours value, T3$, to reflect whether an AM or PM time is being shown. Then line 2250 prints the hours, minutes, seconds and AM/PM marker.

In subroutine 3000, PT is the position of the first colon found in the string T$.

Subroutine 4000 simulates a cursor and constructs T$ from the characters entered through GET A$. No editing is provided, so if you make an error, the entry must be repeated. A little more code could catch A$ = 20 (code for DEL) and give some limited editing (equivalent to back space or rubout on a terminal).

Subroutine 5000 puts the power failure message on the screen and strips the "?" from T$. This permits the display of time code to work correctly.

The astute programmer will note that no provision is made for bad messages from the HP clock (which might make the program fail in some cases). You should check the values T1, T2, T3, T3$, T4$ and T5$ for their legal values and make another attempt to read the time made in case of an error. In the event of several consecutive errors, the program should mention this to the user.

There are limits to how "fail-safe" a program must be made. In many cases, malfunctions will not be critical, and it isn't worth the effort required to make the program survive the errors. I do not recommend the PET for any real-time control applications that may result in injury or loss of property in the event of the PET's failure!

Example 2: The HP 8165A Signal Source

Part 1 introduced the 8165A. Naturally, your interest will be with the devices that you have available, and the example shown here is a "laboratory application"; that is, a program similar to one you might want to build for your instrument.

Let's pretend that the response of a stereo amplifier needs to be tested in a production line. The frequencies and voltages to be tested are:

```
10 Hz,   Sine Wave,    1.000 volts
10 Hz,   Square Wave,  1.000 volts
20 Hz,   .....
20 Hz,   .....
50 Hz,
```

Test sine wave and square wave responses at 1.000 volts for 10, 20, 50, 100 . . . up to 20 kHz.

The plan for a program is as follows:

1) Initialize. For example, open

```
10 PRINT"clr STEREO TEST PROGRAM
20 PRINT"dn dn BE SURE THE 8165 IS ON AND THAT
30 PRINT" THE IEEE 488 IS CONNECTED.
40 PRINT"dn REMEMBER THE ADDRESS FOR THE 8165
50 PRINT"MUST BE 8. PLEASE CHECK THIS.
60 GOSUB 1000
70 OPEN 1,8
80 REM SET UP 8165
90 PRINT#1,"FRQ1ØHZAMP1.ØØØVF1D2OD"
100 REM HOOK UP STEREO
110 PRINT"clr STEREO AMPLIFIER TEST"
120 PRINT"dn ATTACH THE NEW UNIT TO THE
130 PRINT"TEST STATION."
140 GOSUB 1000
200 REM PERFORM TEST
210 PRINT"clr >>>> TEST IN PROGRESS<<<< "
220 FOR L1=1 TO 4
230 FA=1Ø↑L1
240 FOR L2 = 1 TO 3
250 IF L2 = 1 THEN FR=FA/1ØØØ
260 IF L2 = 2 THEN FR=FA*2/1ØØØ
270 IF L2 = 3 THEN FR=FA*5/1ØØØ
275 IF FR >25 THEN 430
280 FOR W = 1 TO 2
290 IF W=1 THEN W$ = "SINE"
300 IF W=2 THEN W$ = "SQUARE"
310 REM SET 8165 UP
320 PRINT#1,"FRQ"STR$(FR)"KHZ"
330 IF W=1 THEN PRINT#1,"F1OE"          (letter F, numeral 1,
340 IF W=2 THEN PRINT#1,"F3OE"           letters OE)
350 REM SET TIMER & REPORT
360 T1 = TI                             (tee one = tee eye)
370 PRINT"hm dn dn da TEST AT:";
380 PRINT"sp sp FREQ:"FR*1ØØØ"sp sp"W$"sp sp sp"
390 IF TI - T1<6ØØ THEN 390
400 REM TURN 8165 OFF
410 PRINT#1,"OD"                        (letters OD)
420 NEXT W
430 NEXT L2
440 NEXT L1
450 REM TEST COMPLETE
460 PRINT"clr ****** TEST COMPLETED ******"
470 PRINT"dn dn REMOVE AMPLIFIER FROM TEST STATION"
480 GOSUB 1000
490 GOTO 110

1000 PRINT"dn dn PRESS ANY KEY WHEN READY"
1010 GETA$:IF A$="" THEN 1010
1020 RETURN
```

Listing 2. Stereo Test program.

the IEEE 488 file.

2) Tell the operator to hook up an amplifier

3) Start the test

4) Loop through the frequencies for each frequency

5) Loop through sine and square

6) Wait for 10 seconds before continuing

7) Report where the test is on the screen

8) End of both loops

9) Tell the operator the test is finished

10) Go to step 2

Listing 2 shows these steps in a BASIC program. From the user's point of view, when the program is RUN, the message below appears:

STEREO TEST PROGRAM

BE SURE THE 8165 IS ON AND THAT THE IEEE 488 IS CONNECTED.

REMEMBER THE ADDRESS FOR THE 8165 MUST BE 8. PLEASE CHECK THIS.

PRESS ANY KEY WHEN READY

This reminder ensures that the 8165 is properly connected, powered and addressed. The PET program won't work if these conditions aren't met.

Now it is time to test a unit. The screen clears (after a key is pressed) and displays:

STEREO AMPLIFIER TEST

ATTACH THE NEW UNIT TO THE TEST STATION.

PRESS ANY KEY WHEN READY

Now the test commences, with a report on the current frequency and waveform being used:

>>>>>TEST IN PROGRESS<<<<<

TEST AT: FREQ: 200 SQUARE (current freq & waveform)

After about two minutes (each frequency and waveform takes ten seconds), the screen clears and tells the user:

•••••TEST COMPLETED•••••

REMOVE AMPLIFIER FROM TEST STATION

PRESS ANY KEY WHEN READY

Now we are ready to perform another test. Look at the scope and notice that the output of the 8165 is turned off between tests and between mounting the new amplifiers. Though unimportant in this example, more serious equipment should always be set to a "safe" state when the operator has to handle the equipment.

Lines 10 to 60 in the BASIC code state the program's name and remind the user to check the address setting on the HP 8165. Subroutine 1000 waits for you to press a key.

Three nested loops are used to scan through the frequencies and waveforms. The L1 loop sets the frequency decade from the range 10-99 Hz to 10000-99999 Hz. The L2 loop is used to select between 1, 2 and 5 times the frequency selected by L1. W chooses between sine and square waves.

Lines 200 to 300 compute the frequency FR in two steps (FA is set to 10^{L1}, and FR is set to 1,2 or 5 times FA), and W$ is set to report sine or square. In line 275 the top value to be tested is 20000 Hz, so to terminate the loops requires a test of the frequency larger than 20000 Hz.

Instead of using 20000 for the test, I am using 25000. (If you look at the code, FA is in kilohertz, so the test is for 25.) Due to the PET's way of computing numbers, when L1 is 3 and L2 is 2, FA turns out to be a tiny amount over 20, which terminates the test too soon.

When testing for equality or differences, make sure the number in the PET is what you think it is. Most floating point numbers will be slightly (and unprintably) different than the value you want, so fudge accordingly.

Line 320 sends the correct command to the 8165 for frequency. Note that FR is sent as the string STR$(FR). This avoids the Cursor Right after the number, which could totally confuse the 8165. Lines 330 and 340 specify the waveshape by directly sending the correct set of characters to the 8165. "OE" turns the 8165 on.

Lines 350 to 390 print the test values and wait for 600 jiffies, or ten seconds. When they are finished, line 410 turns the 8165 off (this is a "safe" state; e.g., during hook-up, the test leads could be shorted).

Lines 450 to 490 announce the end of the test and tell the user to remove the stereo amplifier. Note that the 8165 is in the "off" state.

I will leave it to you to figure out how to use the HP clock to control the timing of the stereo test program (Listing 2, part 2) instead of the PET's internal clock. Another variation is to put up the time each test is run for logging purposes.

More "Gotchas"

Program bugs. When I was debugging the HP Clock program (see Listing 1), the days' count wouldn't come out right. Some months tended to have two or three too many days, while others ran short. For example, May 5 put May 11 on the clock, and February 10 showed February 7.

I first thought that the IEEE 488 device was miscounting characters. I checked by printing the number sent onto the screen. The error wasn't here.

The eventual source of the problem was that the routine that counted the total days in

Function	Old Pet (hex)	(dec)	New PET (hex)	(dec)
Send TALK (MTA)	F0B6	61622	F0B6	61622
Send LISTEN (MLA)	F0BA	61626	F0BA	61626
Send UNTALK	F17A	61818	F17F	61823
Send UNLISTEN	F17E	61822	F183	61827
Set ATN true and send character in accumulator	F0BC	61628	F0BC	61628
Send data character in accumulator••	F0F1	61681	F0EE	61678
Get data character in accumulator	F187	61831	F18C	61836
Flag byte	0222	545	00A5	165

••Set flag byte to FF (255) before calling this routine.

Table 1. PET IEEE ROM and RAM locations.

the previous months just added the same number each time. For May, it added 31 four times, and for February, it added 28 once!

Another bug came from the "hidden bits" in PET numbers. In the Stereo Test program (Listing 2), there was the following line:

```
IF FR>20 THEN....
```

The testing program stopped at 10 kHz instead of 20 kHz. When I printed FR, I got 20. FR was formed from the two computations:

```
FA = 10↑L1
FR = FA*2/1000
```

The PET's exponentiation operator isn't totally exact, so a few bits slipped through. The division didn't help, and FR ended up a slight amount over 20, which is enough to make the condition true. The cure was to test for more than 25 instead.

These errors are subtle. If you aren't a total expert on your PET, these are nearly impossible to find.

```
10 REM PET SERIAL OUTPUT
20 REM GREGORY YOB
30 PT = 826
40 READ BT: IF BT  0 THEN 60
50 POKE PT,BT: PT=PT+1: GOTO 40
60 DIM BD(6),RT(6)
70 FOR J=1 TO 6
80 READ BD(J),RT(J)
90 NEXT J
100 PRINT"clr SERIAL OUTPUT"
110 PRINT"dn PARITY"
120 PRINT"0=EVEN, 1=ODD, 2=MARK"
130 INPUT P
140 IF P=0 THEN 180
150 IF P=1 THEN 180
160 IF P=2 THEN P=255: GOTO 180
170 GOTO 110
180 POKE 994,P
190 PRINT"dn BAUD RATE"
200 INPUT BT
210 FOR J=1 TO 6
220 IF BT=BD(J) THEN 380
230 NEXT J
240 PRINT"RATES ARE:"
250 FOR J=1 TO 6: PRINT BD(J): NEXT
260 GOTO 190
380 POKE 995, RT(J)
390 PRINT"# TIMES TO REPEAT CHAR"
400 INPUT N
410 N=INT(N): IF N< Ø OR N >255 THEN 390
420 PRINT"PRESS ANY KEY TO SEND CHARS"
430 GET A$: IF A$+"" THEN 430
440 PRINT A$
450 POKE 997,N: POKE 992, ASC(A$)
460 SYS(826)
470 GOTO 420

1000 DATA 173,4,2,234,234,240,1
1010 DATA 96,173,64,232,41,64,240
1020 DATA 241,120,21,192,3,144,2
1030 DATA 88,96,32,98,3,32,153
1040 DATA 3,88,76,58,3,234,24
1050 DATA 173,224,3,96,234,169,0
1060 DATA 141,225,3,173,224,3,162
1070 DATA 1,160,0,24,74,144,5
1080 DATA 160,225,238,225,3,72,152
1090 DATA 157,240,3,104,232,224,8
1100 DATA 208,234,273,226,3,48,12
1110 DATA 240,3,238,225,3,173,225
1120 DATA 3,41,1,240,2,169,255
1130 DATA 157,240,3,96,162,255,232
1140 DATA 189,240,3,141,34,232,172
1150 DATA 227,3,173,0,64,173,0
1160 DATA 64,173,0,64,136,208,244
1170 DATA 234,236,228,3,208,228,96
1180 DATA 96,0,0,0,0,0,0
1190 DATA 0,24,173,229,3,208,2
1200 DATA 56,96,173,224,3,206,229
1210 DATA 3,96,0,0,0,0,0
1220 DATA 0,0,0,0,0,0,0
1230 DATA 0,0,0,0,0,65,2
1240 DATA 0,195,11,0,0,0,0
1250 DATA 0,0,0,0,0,0,0
1260 DATA 0,255,0,0,0,0,0
1270 DATA 255,0,255,255,0,0,0
1280 DATA 0,0
1300 DATA -1
1999 REM PARAMETERS FOR BAUD RATES
2000 DATA 9600,5,4800,11,2400,23
2010 DATA 1200,48,600,97,300,195
```

Listing 3. Serial output via the IEEE 488 bus port.

Using the PET ROM

Since the PET knows the IEEE bus, there have to be routines in the PET ROM that know how to work the bus. A year ago, some of my clients' requirements forced me to access the PET's ROM for the IEEE. (One had a machine that didn't like the PET's state between IEEE messages; the other wanted to know the PET's maximum IEEE transfer rate.)

Table 1 indicates the pertinent RAM and ROM locations for the PET IEEE routines. Use caution when working with these, as I have only been able to check the ones mentioned below. In one case, a routine sent a character at an apparent rate of 5000 characters/second! (The listener didn't see anything at all.) The routine in question took a look at the bus, decided the bus wasn't in a legal state and returned, instead of sending the character! If you have an accurate PET disassembly, here is a good place to use it.

Input from the IEEE Bus. This can be approached either from machine language or as a mix of machine language and BASIC. In all cases, the first step is to open a file to the bus via BASIC. (This must be done; make sure that only one file is open.)

The next step is to send a TALK to the device. From BASIC, this is a SYS(61622), and in machine language it is a JSR F0B6 (or 20 B6 F0).

To handshake a character in requires calling the machine language in ROM. Here's a catch: the character arrives in the A register. From BASIC, you must SYS to a short routine that performs JSR F187 and an STA (somewhere) (and RTS to get back). Then PEEK (somewhere) gets your character. The machine code in hexadecimal is 20 87 F1*8D xx xx 60. The xx xx is your "somewhere." The value from the IEEE bus is the complement of your character; that is, the 1's and 0's are exchanged.

Send to the IEEE Bus. Again, the first step is to open a file to the bus and be sure that only one file is open. Then, send the ATN LISTEN via SYS(61626). (In machine language, JSR F0BA, or 20 BA F0.) Now, put the complemented value into location $0222 with a POKE 546, CHARACTER.

The last step is to SYS (61681), which sends the character. In some cases, you will have to set a flag first by setting location $021D to $FF by POKE 541,255. I have used both methods with success.

The machine-language sequence is A9 FF 8D 1D 02 20 xx xx 8D 22 02 20 F1 F0 60. The 20 xx xx is a JSR to your routine at xx xx, which gets a character in the A register.

Both the input and the output will leave the device active on the bus. Make ATN true and send the UNL and UNT value to release the device.

The IEEE lines in the PET don't have to be used for the IEEE 488 bus. There are 12 easily used bits of parallel I/O that can be controlled with suitable PEEKs and POKEs, and two PET Hard Copy Easy," *Kilobaud Microcomputing*, September 1979, p. 100.

Printing Hazards

The difference between the PET's display and character codes and the ASCII character set causes some difficulties when you use the IEEE 488 bus for printed output.

1. ASCII printers use the most significant bit (MSB) as a parity bit. If the PET is sending a graphics character (or lowercase, as provided by the POKE 59468,14 for old PETs), the printer will either ignore this and print the corresponding ASCII for the seven less significant bits or print a "parity error" character. If you get a parity error character, set your printer to the "no parity," or "mark" parity, option.

2. The PET cursor control characters result in the ASCII values in the range 0 to 31, which are control characters in ASCII. If you are lucky, these will be ignored; if you aren't, some of these may result in setting your printer to unwanted modes. (The Comprint printer is

Listing 4. Serial output, machine-language assembly listing.

```
     This code was hand assembled and then patched - so the flow
isn't continuious and there are occasional NOPs that aren't needed.

                              ! Check SHIFT key
033A AD 04 02  SENSE     LDA SHIFT (0203)     read shift key location
     EA EA               NOP, NOP             (tis a patch)
     F0 01               BEQ GO (0342)
     60                  RTS                  back to'BASIC if SHIFT
                                               pressed
                              ! See if device is ready
0342 AD 40 E8  GO        LDA $E840            Get all PB2 lines from VIA
     29 40               AND #40              Mask NRFD bit
     F0 F1               BEQ SENSE (033A)     Wait if not ready

                              ! Set up PET for transmission of characters
                              !  Turn off interrupts
                              !  Get character
                              !   Set carry if no more characters
                              !  Set up Xmission table
                              !  Send character

0349 78                   SEI                  Interrupts off
034A 20 C0 03             JSR FETCH (03C0)     Fetch Character
                                               (Set up as a subroutine
                                                to let you "roll your own"
                                                routine)
     90 02                BCC GO1 (0351)
     58                   CLI                  Interrupts on. If Carry is
     60                   RTS                  set, no more chars to send.
                                               If you make your own FETCH,
                                               use this convention.

0351 20 62 03  GO1       JSR SETUP            Set up Xmit table for char in A
     20 49 03            JSR XMIT             Send char
     58                  CLI                  restore interrupts
     4C 3A 03            JMP SENSE            Look at SHIFT key again
     EA                  NOP                  (patch)

035C 18        FFETCH    CLC                  Dummy version of FETCH
     AD E0 03            LDA CHAR (03E0)      Test Char location
     60                  RTS
     EA                  NOP                  (guess)

                              ! Set up Xmission Table
0362 A9 00     SETUP     LDA #00
     8D E1 03            STA PARITY (03E1)    Initialize parity counter
     AD E0 03            LDA CHAR (03E0)      Get char
     A2 01               LDX #01              X reg counts 7 bits of char.
                              ! Shift char & if carry set, load FF into
                              ! Xmit table. If carry not set, load 00
                              ! (NOTE: Start & Stop bits are assumed preset
                              ! in Xmit table. Be sure this is so in your
                              ! program too.)

036C A0 00     SLOAD     LDY #00              Y holds 00 or FF for bit
     18                  CLC                    in char.
     4A                  LSR A                Shift LSB into Carry
     90 05               BCC HOPPITY          Bit is zero
     EE E1 03            INC PARITY (03E1)    '1' bit adds to parity count
     48                  PHA                  Stuff A on stack
     98                  TYA                  Y to A
     9D F0 03            STA START,X          Put into Xmit table. I just
                                              love non-symmetrical
                                              instruction sets! (6502
                                              has no Y indexed addressing)
     68                  PLA                  Restore A from stack
     E8                  INX                  On to next bit
     E0 08               CPX #08              7 bits yet?
     D0 EA               BNE SLOAD (036C)     no, repeat

                              ! According to  option, set the parity
                              ! bit in the Xmit table

0382 AD E2 03               LDA POPTION (03E2)  Get option value
     30 0C                  BMI MARK            MSB means MARK parity
     F0 03                  BEQ EVEN            zero is EVEN
     EE E1 03               INC PARITY          Add 1 for odd parity
     AD E1 03  EVEN         LDA PARITY
     29 01                  AND #01             LSB has parity in it
     F0 02                  BEQ ZILCH           Save LDA #00 if A is 00
     A9 FF     MARK         LDA #FF
     9D F0 03  ZILCH        STA START,X         Put in Xmit table. X happens
     60                     RTS                 to be right value!

                              ! Send Character
0399 A2 FF     XMIT      LDX #FF              The next instruction
     E8        CONT      INX                     makes X zero.
     BD F0 03            LDA START,X          Get byte to send
     8D 22 E8            STA $E822            Put on IEEE DIO Lines (out)

                              ! Delay loop for baud rate
03A2 AC E3 03               LDY RATE (03E3)     Get countdown value
03A5 AD 00 40  AGAIN        LDA $0400           4 cycles of delay
     AD 00 40               LDA $0400              ditto
03AB AD 00 40               LDA $0400              ditto
```

a "lucky" one.)

3. As a result of these first two steps, if you use CMD and LIST, the listings you get will have missing or misleading characters. I have a program (drop me a card) that will list a BASIC program in a legible form.

4. The PET does not transmit a line feed. You must provide CHR$(10) after every carriage return.

5. If your printer needs a carriage return delay, either print the required number of CHR$(0) or insert a small waiting loop; i.e., FORJ = 1TO20:NEXT.

6. Most printers have no formatting capabilities. If you keep careful count of the number of characters sent, formatting is clumsy, but possible. Pitfalls include:

- A printed number has a CHR$(29) sent after the last digit, which is not a space and is usually ignored by printers.
- TAB and SPC provide CHR$(29), and not spaces. (New PETs have this fixed.)
- LEN(STR$(number)) will not count a CHR$(29), since STR$ produces a string without a blank or skip after the last digit.
- If the number is small or large, beware of scientific format; i.e., 1.23E+23.

7. If you are attempting a word-processing program, the PET's codes for the lowercase characters and the ASCII codes are different. The PET thinks the lowercase letters lie in the range 192 to 223, and ASCII likes the range 96 to 127.

A further complication is that the ASCII character set and the PET character sets don't match. Backarrow on the PET is ASCII underline; the curly brackets, vertical bar and tilde in ASCII don't exist on the PET. The ASCII accent mark (looks like a reverse apostrophe) is seen by the PET as a space. Your printer might have other character options to puzzle you.

Wrapping It Up

Working with the IEEE 488 bus is nearly an entire engineering discipline in itself. I hope my efforts enable you to get

```
     88                 DEY                 reduce countdown
     D0 E4              BNE AGAIN (03A5)     keep going till count is zero
     EA                 NOP                 Successful branch takes 3
                                             so this compensates to
                                             make a 17 cycle per loop
                                             delay
      EC E4 03           CPX BITCOUNT        Check number of bits to
                                              be sent.
      D0 E4              BNE CONT            Do next bit
      60                 RTS

.......(some room here) .......

                         ! Fetch Character for real. Feel free to
                         ! make your own routine. Set carry bit when
                         ! out of characters.
03C0 18        FETCH     CLC                 Be sure to do this!
     AD E5 03            LDA CHCOUNT (03E5)  # chars to send
     D0 02               BNE OK
     38                  SEC                 Set  carry, out of chars
     60                  RTS

     AD E0 03  OK        LDA CHAR            Get char - you might use
                                             TAX & LDA CHAR,X here.
     CE E5 03            DEC CHCOUNT         decmt chars counter
     60                  RTS

....... (some room here) .......

                         ! Data Area

03E0 00        CHAR      ! Character to send. (Move elsewhere if you
                           want to send more than one)
03E1 00        PARITY    ! Parity Counter
03E2 00        POPTION   ! Parity Option. 0-even,1-odd,FF-mark
03E3 00        RATE      ! Initial countdown for baud rate. POKEd
                           by the BASIC program.
03E4 00        BITCOUNT  ! Number of bits to send (10 or 11 decimal)
03E5 00        CHCOUNT   ! Number of chars to send

....... (a gap again) ......

03F0 00        START     ! Start of Xmit table
03F1 00 00 00 00 00 00 00     ! Character, lsb first
03F8 00                  ! Parity bit
03F9 FF FF               ! Stop bit(s)
```

aboard the IEEE 488 bus of your PET and turn it to some profitable use. ■

References

1. "IEEE Standard Digital Interface for Programmable Instrumentation," IEEE Std 488-1975, ANSI MC 1.1-1975.

2. Hewlett-Packard, 1502 Page Mill Road, Palo Alto, CA or PO Box 301, Loveland, CO 80537. Several publications are available on request.

3. "PET 2001-8 User's Manual" and "PET Communication with the Outside World," Commodore Business Machines.

4. "Getting Aboard the 488-1975 Bus," Motorola.

5. "PET User Notes," PO Box 371, Montgomeryville, PA 18936.

6. MOS Technology, Inc., 950 Rittenhouse Road, Norristown, PA 19401.

THE 8000 SERIES— OR SUPERPET

Dave Middleton

The long awaited 80 column screen PET is now available from Commodore Commercial Sytems dealers around the country. The 8000 series maintain compatibility with the 3000 series for most applictions where BASIC is used.

The 8000 series support what is called BASIC 4.0 or PET disk BASIC. 14 extra commands have been added which allow far easier communication with the disk unit. There are also two reserved variables DS and DS$ which are used to interrogate the disk system for error messages. For instance DLOAD"PROG-NAME" will load 'PROG-NAME' into RAM ready to be RUN.

The disk commands are only recognised by DOS2 which is in the 8050 1 megabyte disk drive but the chips will be available for retrofit in the near future from your dealer. Note also that a special version of BASIC4 is being produced for the 3000 series PET so that the advanced features will be available to all users.

The other new item for the 8000 series is of course the screen. This has many advanced features apart from being 80 characters wide. There are for instance two mode of operation, text mode where there is small gap between the lines making text easier to read and graphics mode where the lines are closed up as with the 3000 series. A simple command allows easy switching between the two. It is also possible to define a 'window' in which all text is to be printed, leaving the rest of the screen untouched; once again a couple of simple commands allow a window of any size to be formed on the screen. The screen RAM is now 2000 characters (80x25 lines) and this may have an effect on some programs which address screen loctions with PEEK/POKE commands.

Every 2nd line on the 40 column screen is now at the right hand side of the 80 column, so displays which use actual screen locations such as POKE33768,160 will look very strange when transfered directly to the 8000 series from the 3000 series.

Another advancement is in the garbage collection routine which could 'hang' the PET for up to 10 minutes while it removed rubbish from string storage. Also it is now possible to use repeat on all keys so there is no need to, for instance, keep stabbing at the cursor control keys to get to a specific point on the screen. Anybody who has tried editing a program will know how frustrating it can be trying to get out of 'quotes' (") mode to use the cursor keys well all you have to do with the 8000 series is press the 'ESC' key and you can continue to edit the line as if nothing had been changed.

The 8000 series has a standard QWERTY keyboard which will be a great relief to a lot of experienced secretaries who must have been greatly perturbed by the 3000 series with the full stop on the numeric keypad.

Remember how any program which used zero page memory in the Old and New ROM 3000 series had to be carefully modified? BASIC 4.0 maintains the same zero page addresses as the New ROMs for the 3000. The only changes being in the screen wrap for cursor control to allow the 40 column PET to have 80 column BASIC lines. These location are superfluous to the 8000 series and are used for, amongst other things, controlling the window.

OWNERS REPORT

Mike Todd

Mike's PET was one of the first 8K machines to arrive in the U.K., and the fault to which he refers (caused by frequent opening and closing of the PET casing) has not been reported for many months. In spite of early setbacks Mike became involved in his PET and developed some interesting techniques which are reproduced here. He has recently had his machine fitted with the new ROMs and his latest discoveries will be published in a future article.

I've had my PET for over 18 months, and in that time I have had more than my fair share of problems - but in that time I have also come to understand many of the PET's idiosyncracies. Could all of my problems have been due to the fact that it was delivered on March 13th 1978 at 1.13 pm (1313 on the 24 hour clock) or that I was living in London W13 in a house number 52 (= 4 x 13) or even that its serial number is 1000013!!? I wonder....

Some of the most aggravating problems have been fairly simple, mainly due to dirty connectors. These faults included very severe screen judder, intermittent

keyboard operation, intermittent cassette reading and writing, even the PET crashing completely - all were cleared by careful cleaning of the internal connectors. I would certainly recommend periodic cleaning of these connectors (say once every six to eight months) with a commercial switch cleaner. The most important to keep clean is the main power connector (that's the one which comes from the mains transformer area), since this can put a high resistance path in the power supply, resulting in poor regulation and 'glitches' on the +5 volt rail. However, DON'T be tempted to clean the IC sockets - this would be asking for trouble! I would also add that, if your PET is transported a great deal, the connectors can work loose and may need the occasional push home.

Some models (like mine) have a minor fault which allows the cassette bracket to scrape on the low power lead to the logic board, eventually scraping away the insulation and shorting the supply to chassis. I can only suggest that this connector is bent very slightly backwards (very carefully) and the cassette bracket is pushed forward to keep it clear. A piece of insulating tape around the wires where they enter the plug may also help.

Surprisingly, I have little problem with the cassette unit, such as the reported partial erasure of tapes and azimuth errors, and can only attribute this to the careful cleaning of the heads now and again, as well as a total lack of tampering with the mechanics of the recorder.

The remainder of this article describes some of the software techniques that I have found useful, but I must point out that they are designed to work on the 8K machine, and therefore may not work on the 16/32K machines or on 8K machines with the new ROMS. If you are in any doubt, just try the routines out - you can't do any damage. If they don't work, then the routines and memory locations may have to be translated into the addresses appropriate to the machine. I hope to do this myself very soon.

If you are bold enough to write machine language programs then you will appreciate the waste of time and space that DATA statements containing the machine code can take up. If your program resides in the second cassette buffer then it is possible to SAVE a composite version of your machine code and BASIC program. During development it is better to keep the two programs separate which will allow you to work on each program independently of the other. However, when development is complete it is easy to SAVE a composite version.

First, LOAD the machine code program into the second cassette buffer, then clear memory by typing NEW and LOAD the BASIC program. All you need to do is to type the following :

```
A=PEEK(124):B=PEEK(125):C=number of
characters in program name

POKE249,0:POKE250,128:POKE229,A:
POKE230,B

POKE247,58:POKE248,3:POKE238,C
```

Now clear the screen and type the name of the program at the top of the screen keeping the shift key depressed. You'll get a load of gibberish if you are in graphics mode, but don't worry. Move the cursor down the screen a couple of lines and type SYS(63153) and the composite program will be SAVEd - from the start of the machine code to the end of the BASIC program.

This program may then be LOADed and RUN in the normal way without any need for statements in the BASIC program to load the machine code program.

It might be worth adding that this technique will only work if the PET has already LOADed a program from cassette 1.

I have read a lot about the problems associated with leaving FOR...NEXT loops before the NEXT is encountered - this can cause problems because the FOR entry is left on the stack and can use up space until it is cleared by the final NEXT in the loop. However PET BASIC is rather more intelligent than it is given credit for. Any open FOR...NEXT loops appearing in a subroutine are closed, and their entries on the stack cleared, as soon as the RETURN is encountered. This is done because the GOSUB entry on the stack will occur before the FOR entries, and as soon as the RETURN is encountered the stack is cleared up to and including the GOSUB entry. In addition, whenever a new FOR...NEXT loop is encountered, the stack is purged of any open FOR entries using the same variable - thus two FOR...NEXT loops using the same variable cannot be open at the same time. Thus, leaving open FOR...NEXT loops is very rarely a problem provided that these points are borne in mind.

One of the greatest problems with the PET is the inability to communicate directly with your own output routines - the problem occuring when trying to pass the parameters to the routine. The most common method is to convert characters to numeric values using BASIC statements, and then pass these to an output routine using the USR(X) function. However, it is possible to pass parameters after any expression (string or numeric) and pass the result to the output routine. Thus, if an output routine were sitting in the second cassette buffer, SYS(826)A$ would pass the string A$ to the routine while SYS(826)4.3*A would pass the value of 4.3*A. The method is very straightforward and has many applications other than for output routines.

In the interpreter there is a routine (at CCB8) which will scan an expression,

evaluate it and determine if the result is a string or a numeric value - if it is a string, the length and starting location of the string is stored; if it is numeric, the result is placed in the floating point accumulator (FAC), and this can be converted to a string by using two more routines (DCAF & D36B). Finally, loose ends must be cleared up and the string pointers gathered together - this is done by routine D57E.

When all this is done, 0071 and 0072 hold the pointer to the start of the string and the accumulator holds the string length. Therefore it is easy to extract characters one at a time from the string and output them as appropriate. The following example illustrates the technique using the PET output routine at FFD2 which will output a character to the screen from the accumulator.

```
033A 20 B8 CC JSR CCB8     ; evaluate
                             expression
033D 24 5E    BIT 005E     ; test for
                             result type
033F 30 06    BMI 0347     ; branch if
                             string
                             variable
0341 20 AF DC JSR DCAFA    ; convert
                             FAC to
                             string
0344 20 6B D3 JSR D36B     ; get string
                             pointers
0347 20 7E D5 JSR D57E     ; clean up
                             pointers
                             etc.
034A A0 00    LDY #00      ; clear out-
                             put pointer
034C AA       TAX          ; X holds
                             number of
                             characters
034D F0 09    BEQ 0358     ; branch if
                             none
034F B1 71    LDA (0071)+Y; fetch char-
                             acter
0351 20 D2 FF JSR FFD2     ; and output
                             it
0354 C8       INY          ; increment
                             output
                             pointer
0355 CA       DEX          ; count char-
                             aters
0356 D0 F7    BNE 034F     ; branch if
                             more
0358 60       RTS          ; return
                             when
                             finished
```

This routine acts in a similar way to PRINT, but does not allow formatting controls (e.g.TAB, comma etc.) or more than one expression. The instruction JSR FFD2 could be replaced by any output routine, e.g. to drive a Selectric, or (as I use it) to drive a Prestel/Viewdata television which gives FULL COLOUR GRAPHICS and alphanumerics!! The majority of ASCII control characters can be passed using reverse field characters (e.g. reverse field 'A' return a value of 1 etc.); however, beware of using reverse field 'M' or 'O' since these can produce some odd results.

The use of reverse field characters allows the user to include DELete (reverse field 'T') and carriage return (reverse field shifted 'M') into strings. Whenever the character is encountered it is treated as a control character, even when LISTing - thus a line which ends with a string containing several reverse field 'T' will DELete itself on LISTing!! - this would allow various parameters to be 'hidden' from the general user.

Returning to machine code output routine; if your routine writes directly to the screen, you will be well aware of the snow effects that can be encountered. There are two ways of overcoming the effect; the first is to turn off the screen while writing (the PET does this every time it scrolls the screen) using the instructions LDA#34 STA E811, replacing the 34 with 3C will turn the screen back on again; the second is to force the program to wait until the video driver is not accessing the video RAM, (i.e. during vertical retrace) by waiting for the SYNC signal to go low. This can be done using the following routine which should immediately precede the output instruction:

```
WAIT  LDA E840   ; get ORB
      AND #20    ; isolate SYNC bit
      BNE WAIT   ; loop until SYNC
                   bit clear
```

This can be disabled by POKE 59458,44 which forces SYNC low so that it always appears to be in a retrace condition; POKE 59458,12 will restore operation to normal. Notice that the PET uses this waiting technique when outputting to screen and the disabling instruction can be used to speed up all PET output (program output and LISTing).

For those who output to screen in BASIC, WAIT 59456,32,32 has a similar effect in reducing snow at the cost of reduced output speed.

For those who are ambitious enough to make a very minor modification to the PET, I can recommend the inclusion of a small earpiece insert (of the type found in telephones - usually available through electronic junk shops) connected in series with a 20Kohm resistor across the cassette 1 'read' line. This provides useful information as to what the cassette is doing, and gives some idea of where you are on the tape. The addition of this has no adverse effect on the operation of the machine provided that the d.c. resistance is at least 20Kohms. There is no reason why the earpiece should not be made switchable between the cassette line and one of the PET outputs so that PET music fiends can use it as a built-in loudspeaker. With practice it is possible to identify leader, header blocks, program blocks and data blocks as well as interblock spacer. It also allows verification that the cassette read/write is functioning.

For those who program in BASIC, it is often necessary to convert from decimal to hex (or some other base) and back again. The following one-liners will do this fairly efficiently:

DECIMAL to HEX (nb: D contains decimal number and H$ must be clear to "" before entry; H$ holds hex value on exit).

```
10 IF D THEN A=INT(D/16) : H$=MID$
   ("0123456789ABCDEF", 1+D-A*16, 1)
   +H$:D=A: GOTO 10
```

HEX to DECIMAL (nb: H$ contains hex number, D exits with decimal value).

```
10 D=0 : IF H$>"" THEN FOR I = 1 TO
   LEN(H$): A=ASC(MID$(H$,I,1))-48
   :D=D*16+A+(A>9)*7 : NEXT
```

These routines can be changed for any base merely by changing every occurence of 16 to the new base.

Many of you may have heard or read of the new improved 'PET user manual' which has just been published. I managed to buy a copy some time ago and must say that I was very pleasantly surprised.

The new handbook (there appears to be a different one for each of the available machines) contains over 120 pages covering the basic operation of the PET, including detailed descriptions of what the interpreter is up to. For instance did you know that whenever FRE(0) is encountered, the PET does a massive garbage collection routine which clears up all string variables not being used, and makes as much space available as possible? Coverage is given to all aspects of BASIC programming together with a couple of example programs. There is also a section on use of the PET input/output ports, details of the IEEE bus and a complete listing of the machine language monitor which is available on cassette.

The appendix contains details of error messages, a detailed memory map (including page zero), details of variable storage and a reference list of BASIC statements, unfortunately many items are omitted from the list (such as string operators) which makes the reference incomplete. There is also plenty of space for your own notes. With the final appendices covering how to conserve time and space, a parts list and some suggested reading (USA oriented), I can recommend this handbook as essential to anyone who takes PET programming seriously. I might also add that this is unsolicited testimonial!!

Finally, going back to the plethora of '13's in the life of my PET, I was shocked to discover that my PET was 13 months old on Friday 13th April 1979!! I wasn't superstitious; but now, well.....

PASCAL REVIEW
Rob Evans

The following article was written by Rob Evans and will give an insight into the new Pascal compiler for the PET. A full description of what the compiler is capable of is impossible to give in a few lines so Rob has detailed the differences between Commodore Pascal and that laid down in the 'Pascal user manual and report' by Jensen and Wirth. There is a second article giving details of an actual Pascal program written by Rob based on the game Mastermind, this will be published in CPUCN 3.1

Keith Frewin left University College London with a 1st Class Honours degree in Electronic Engineering and Computer Science and started work with Transam Computers, a year later he produced the first Pascal compiler to run in under 32k of RAM for the Triton micro-computer. Commodore, interested in another powerful language approached Transam and 4 months later the Pascal compiler for the PET made its appearance. There are very few differences between the two versions of the compiler; provision had to be made for a different disk operating system and the slightly lower memory available on the PET but the compiler accesses the PETs floating point package in ROM thus keeping the 9 digit accuracy for high precision arithmetic.

Large portions of the compiler were written in Pascal with machine code being used where speed is of high importance hence Keith only had to change the machine code sections to get the PET version running, thus showing the great portability of Pascal.

Keith is now in the process of converting the compiler to run on the 80 column PET.

Before giving you my review of Pascal I have been asked to say a few words about myself.

Although at present I work for Commodore's software department I am, as it were, on loan from Brunel University for a period of 22 weeks. The reason being I take a computer-science undergraduate sandwich course, which includes 3 industrial training periods of 22 weeks and Commodore kindly employed me for the second of these (incidentally anyone interested in employing a Brunel student in a similar capacity next year can contact me at Commodore now!).

A large amount of my time at Commodore has been spent testing the Pascal system and editing the manual supplied with it -

and a most enjoyable job too! However all good things must come to an end and Pascal went onto the market in mid-July. Incidentally we produced 1,000 copies of Pascal on the first production run some of which have gone abroad, as we are confident it will prove very popular.

Anyone interested in name derivations may know that Pascal was named after the French mathematician Blaise Pascal. Who according to 'Who Did What' invented a calculating machine back in the 17th century and was presumably thus honoured. But if Pascal were an acronym then how about Programming with A Structured Concise Algorithmic Language - any better suggestions?

By the way we shall soon be starting a Pascal user's group, initially as a couple of pages in CPUCN, anyone interested in contributing anything or in helping to answer the questions from other Pascal users please write to the editor at Commodore Slough.

The hardware required to develop and run Pascal is the CBM Professional computer with 32K RAM and BASIC 2.0, plus a model 3040 floppy disk unit with DOS 1.0.

Firstly and perhaps most important is the cost, which is 120 pounds, for this you get the Pascal development system on diskette plus a comprehensive 100 page manual which comprises of 3 chapters; Introduction to Pascal; Beginner's Guide to Pascal; and the Pascal Reference Manual.

At the outset let me say that this version of Pascal contains all the features included in the 'Pascal User Manual and Report' by Jensen and Wirth, as well as many additional features which will be subsequently described.

The Pascal system comprises a number of programs and files which are needed to develop and run your programs. Tackling them in the order you would use them when developing a program the first is the EDITOR (4k). The editor exists to allow Pascal programs to be keyed in and saved under a file name with the greatest convenience to the programmer. The Pascal source program written in pseudo English (i.e. Pascal!) while convenient for you is not understood by the PET which only understands it's own language - machine code. So the second program provided is the COMPILER (14k) which converts your Pascal source program into a Pascal object program - know as P-code. P-code is an intermediate code between source code and machine code and so requires another program, the INTERPRETER (10k), to read the P-code and execute it.

However starting at the beginning, Pascal is loaded as any other program would be and from this point on you are in Pascal resident mode, although it is easy to switch in and out of BASIC as you shall subsequently see. Resident refers to the Pascal compiler, which is resident in RAM, and is distinct from the disk mode when the compiler is held on disk (until required) - disk mode allows an extra 24k for your source program, 28k in all. - In resident mode you are operating in a similar mode to what you might be used to with BASIC. Each statement is given a line number and when all the program is keyed in, it is executed by simply typing RUN. All Pascal statements are available in resident mode except disk commands and obviously there is a restriction as to the size of the source program. So summing up, resident mode facilitates the fast development of smaller, simpler Pascal programs and primarily is envisaged as an aid to becoming proficient in Pascal very quickly. When you write your programs for real then you will use disk mode.

In disk mode, you can write larger programs because the compiler resides on disk. Also in this mode ALL features of the Pascal system are available. Disk mode operates in a similar fashion as when developing programs on a mini or main-frame computer. When a Pascal source program is created via the editor instead of immediately executing, it is saved as a file on disk to be subsequently compiled and stored as an object or P-code file. Upon execution of the COMPILE COMMAND, the compiler is read into RAM and the source file read from disk, compiled and written back to disk as a new (object) file under the same name as the source, but with the suffix '.obj'. To execute this program the EXECUTE COMMAND reads the interpreter into RAM and the interpreter reads the program object file from disk and executes it.

Common to both modes of operation is the EDITOR - the program which allows you to enter Pascal source statements. It is based on the BASIC editor but with the following additional features:-

1) Automatic line number generation after the first line is entered.
2) The ability to specify either upper or lower case display. On a personal note I think lower case is infinitely more presentable and in fact you are automatically in lower case when you load Pascal.
3) BASIC may be entered any time simply by typing BASIC and then you may revert back by typing PASCAL.
4) BREAK allows you to enter the machine code monitor.
5) DISK allows you to enter disk mode from resident and RESIDENT switches you back again.
6) Renumbering is achieved with the NUMBER command.
7) Very useful extras are the FIND, CHANGE, and DELETE commands, all of which may be specified within a range.
8) To run programs in RESIDENT MODE use R or RUN, L, or P, where L lists the

program on the screen and P on the printer.

9)In DISK MODE file transfer between PET and disk is achieved with the commands GET and PUT.

10)To compile and execute a program in DISK MODE, COMP and EX are available.

11)The HEX and DECIMAL commands are very useful giving instant conversion between the two bases.

12)Finally the LINK command creates one large object module from independently compiled modules as described below.

In addition to the standard features of Pascal there are a number of extra's described below, included for the programmer's convenience. These are:-

1)The use of HEXADECIMAL CONSTANTS.

2)PEEK and POKE.

3)A function, called 'ORIGIN', to allow pointers to be set to a physical memory location. For those new to Pascal a pointer is a variable which points to the address of a variable rather than holding the actual value. The use of which is beyond the scope of a review!

4)A funtion, called VDU, whose arguments accept screen row and column numbers and the character to be displayed thereat. This function is very useful for displaying patterns on the screen!

5)HEXADECIMAL INPUT/OUTPUT is allowed.

6)There are 6 bit-manipulation functions. For example, ANDB accepts 2 arguments and where corresponding bits are set outputs '1' else '0'.

7)You may specify whether input/output errors - for example a character is keyed-in instead of an integer - will cause the system to abort with a suitable error message or simply allow the program to continue.

8)The STOP key may be enabled/disabled as required.

9)A RANDOM NUMBER GENERATOR produces a random number in the range 0-255 and can be manipulated to generate a number in the range 0 to MAXINT.

10)An underscore character can be used to present variable names more neatly (by the way variable names can be up to 16 characters!) for example, NEW COUNT.

11)The PET internal clock may be both set and read.

12)String variables may be input. This is extremely useful for inputting file names.

13)And finally program CHAINING is allowed, as described below.

Anyone concerned about program size restrictions will be pleased to know that careful program design using either CHAINING or LINKING (or both) will in most cases solve this problem.

LINKING allows independently compiled modules to be combined and approximately 3,000 source lines may be thus linked in this fashion. A linked program contains a main-logic module and a number of procedure modules. Each module is compiled independently. Procedures may be called by the main-logic or other procedures. When a procedure is called within a program program-control passes to it and on exit from the procedure returns to the statement immediately after the procedure call. There are many benefits to be had designing programs modularly. For example, the development of each module is isolated from the rest allowing concentration on logically separate sections of the program. Variables common to ALL modules may be specified by declaring the same list of variables at the front of each module. And this is easily achieved by creating a file of variables and using the FILE APPEND command to append the variables at the front of each module. This is extremely neat in practise because if a variable changes then it is simply changed once in the file of variables. Also debugging is made easier and a recent method I adopted was to display the module name upon entry. Program bugs are then isolated to one module which may be efficiently amended, compiled and relinked with the remaining modules. I would expect most decent-sized programs to be created using linking, which I hope you will find a very effective method after an initial period of familiarisation.

CHAINING allows you to leave one program and enter another from within your Pascal program. Also if the variables are specified as being the same in both programs (or however many are chained) then the variable's value will be maintained from one program to the next. Using chaining a suite of programs may be run without operator intervention.

Too numerous to list are all the arithmetic and conversion funtions but they include the following; sine; natural logarithm and odd (which returns 'true' if the argument is odd and vice versa).

Finally I do not pretend that such a review could pre-empt all possible questions you may have about Pascal, so please do not hesitate to write to the Pascal user group at Commodore Slough should you require further information.

Following on from the REVIEW of Pascal in the last newsletter this article about Pascal is divided into three sections. The first, 'who will use Pascal', outlines exactly who I think will benefit from Pascal. The second section, 'how to learn Pascal', attempts to explain how you approach the problem of learning Pascal rather than actually trying to teach any syntax. This is because once you accept the structure of Pascal actually learning the syntax is relatively simple. The last section simply aims to get you familiar with PET Pascal.

WHO WILL USE PASCAL

There are many people who may consider using PET Pascal, including,the lecturer thinking of teaching Pascal to students; a writer for a software-house; or a hobbyist moving 'up market'. If you need reassurance of this look no further than the American Department of Defence (DOD) who are pouring oceans of money into developing ADA - a language which is based on Pascal!

The University lecturer thinking of teaching Pascal on the PET will no doubt be aware of the general advantages of teaching Pascal on a microprocessor over teaching Pascal (or any other language) on a mainframe-computer - which in a nutshell is 'user-friendliness'. As a student who has recently undergone both methods of learning a new language (ALGOL68 on a mainframe-computer and Pascal on the PET) I have no hesitation giving my vote everytime to the PET. We have sold 25 copies of Pascal to Strathclyde University who intend teaching Pascal to 250 first-years and more advanced features to computer science second-years. Strathcylde exhaustively tested Pascal before they invested so heavily in it and we regard their faith in PET Pascal as a tremendous compliment. Who knows they might even produce a 'Strathclyde Pascal', along the lines of 'Strathclyde BASIC'! Which as most of you will know is a training-kit comprising programs and manual. We have also had enquiries from UWIST, Cambridge University and Chichester College and I'm hoping my own University, Brunel, will use PET Pascal.

A software-house should weigh the advantages and disadvantages of Pascal to BASIC. Advantages are many and include the following:-

1. Programs which are easier to write and debug because they are split into modules.
2. Programs which are easier to maintain because Pascal is easier to understand than BASIC.
3. Structured data types allowing, for example, lists and records.
4. Quicker program execution time (about 2 or 3 times quicker in the general case).
5. Immense satisfaction from writing in a structured-language.
6. Finally the hope that the more professional programmer will be attracted by Pascal.

No doubt the list the could be greatly extended but I've listed what I consider the most important advantages. One disadvantage with Pascal vis-a-vis BASIC is that strings are not so well catered for and must be manipulated via arrays. Software-houses that write some routines in machine-code note that they can still enter these routines from within a Pascal program.

Turning to the hobbyist I admit 120 pounds may seem a lot to spend in one go, but this must be offset with the many hours of pleasure I hope you will receive and dare I say prestige associated with being a Pascal user. Incidentally for a new language this is not considered expensive, a view tacitly shared I'm sure by those reacting with a thoughtful nod, when we mentioned the price at recent exhibitions.

HOW TO LEARN PASCAL

Anyone who already knows a block-structured language such as ALGOL or CORAL will find Pascal a delightfully simple language to learn. It is very straight-forward and neat. Because of this those approaching block-structured languages for the first time could not have chosen a better language and should experience none of the frustrations associated with similar languages such as ALGOL68, which is extremely complex.

For those outside the educational environment who have decided to teach themselves Pascal be, open-minded in your approach. The reason I mention this is that when switching from BASIC to Pascal one enters a new and strange programming world, familiarity with which will be richly rewarded with better and more satisfying programs.

If you are to teach yourself Pascal then you need look no further than the manual supplied with your copy of Pascal. In my opinion it sets a new high standard for manuals. It contains a chapter for beginners to Pascal and has 101 pages. Useful are the numerous example programs included where new features of the language are introduced. As supplementary reading I can personally recommend 'Pascal User Manual and Report' by Jensen and Wirth, published by Springer-Verlag, which costs £5.95. After an initial, hopefully painless, period familiarising yourself with the concepts of block-structured languages the transition from BASIC to Pascal programming should be quite smooth and I am confident that in the long run the rewards, both in terms of professionalism and satisfaction, far outstrip the initial effort.

As a final note on learning Pascal I would hope that the Training Department will offer a course in the near future, so watch out for announcements.

PRACTICAL EXPERIENCES OF DEVELOPING A PASCAL PROGRAM

As an insight into program development I'll briefly explain the method I adopted developing a program to play a word game I know as Jotto or bulls-and-cows (with no bulls only cows!). The game is not unlike a word mastermind type game which

uses words of 5 letters and the program knows 1926 words. I use a 1926x5 character array which indicates the size of array available in Pascal. After deciding the logic the program was divided into a main-logic section and a number of procedures. The main-logic therefore consists of Pascal statements and procedure calls. To facilitate communication between main-logic and procedures a common variable list was used. This list was created as a file (logically enough called variables!) and subsequently appended to the front of each individually compiled module i.e. main-logic and procedures. The appendage is achieved in one short line:

```
#filename
```

So if a variable changes in any way you only have to alter one set of variables and simply recompile each module.

To illustrate a Pascal program you might like to study the following Pascal source lines:-

```
10 program screenprint;
20 var i,j:integers;
30 begin
40   page;
50   for i:=0 to 24 do
60      for j:=0 to 39 do vdu(i,j,'*')
70 end.
```

The first thing to notice is the layout of the Pascal source lines. Not each line starts in the first column. All Pascal programs are written in this fashion to reflect the structure of the program and the above program may equally have been written (remember it's on a 40 column PET):-

```
10 program screenprint;var i,j:integer;b
egin page; for i:=0 to 24 do for j:=0 to
 20 39 do vdu(i,j,'*') end.
```

Let me explain. All lines of code starting in the same column will be executed the same number of times, and this number may be from 0 to infinity. To help you there are THREE types of loop in Pascal, one of these, the 'for' loop you will have used in BASIC. If lines of code starting in the same column (or blocks of code) are contained in other blocks then they will be repeated each time the higher block is repeated. So back to our example the 'begin' on line 30 and the 'end' on line 70 brackets our program block, all code within the 'begin' and 'end' is executed once. And reading from the top, 'page' is executed first, where 'page' is a handy Pascal function which clears the screen. Next, on line 50, we find the first 'for' loop and this is executed 25 times, however EACH time this loop is executed the subordinate 'for' loop is executed 40 times. The function 'vdu' writes the given character to the screen coordinate, so the first time it is executed '*' is written to the top left hand corner of the screen. The program therefore clears the screen and then fills the screen with '*' row at a time. Also you may have noticed that the two integer variables i and j had to be declared before use.

The advantage of compiling each module separately is that you can concentrate on logically independent and hopefully small sections of code at a time. These can be clean compiled before progressing to the next.

When a module is clean compiled Pascal produces a second file, of the same name, but with the suffix '.obj'. For example, a module called 'main' becomes 'main.obj'. This file, known as the object, contains the P-code for that module, and it is this P-code file that is subsequently interpreted at run time.

However before the program can be executed each constituent module must be clean compiled and the whole lot tied together by using the link command. The link command requires that you specify program name and constituent modules and outputs the modules as one big file which is thus labelled with the program name. This file is the one that is used when running the program.

By the way, to save typing effort store the link command as a file then simply recall this file and edit out the line number each time you execute a link. This is a good save as module-name's tend to be long and tedious to type. Also don't make link commands too long as they must fit into 2 lines i.e. 80 columns. I made this mistake and found it a nuisance to recompile the modules with shorter names! Once the program has been successfully linked it can now be tested for logic errors i.e. debugged!

As a simple debugging aid displaying the procedure name upon entry into that procedure was extremely effective. This idea can be extended further by displaying pertinent variables at strategic places within procedures. Proceeding in this fashion bugs can be isolated to a particular module, which is amended, recompiled, and then relinked with the remaining modules before commencing the debugging procedure. For closer scrutiny of source code a printed listing can be obtained by setting a flag when compiling. Continuing in this fashion I found the program was developed reasonably quickly (You may even see the finished product if Commodore decide to release it!) and also with frequent interruptions working in a busy office I found the method of dividing the program into modules extremely effective allowing one to concentrate on small sections of code.

USER'S CLUB

As I mentioned in the REVIEW of Pascal - see CPUCN vol. 2 no. 8 - we hope to be starting a Pascal User's Group, which

would communicate via the CPUCN. Anyone interested in sharing their experiences of PET Pascal, whether they are programming tips or whatever should contact Dave Middleton at Commodore Slough.

THE BASIC PROGRAMMERS TOOLKIT—A REVIEW

The BASIC Programmers toolkit is a piece of firmware, developed specifically for the PET, in the States, and imported into this country by PETSOFT. Essentially the toolkit adds 10 commands to the PET operating system by means of a PROM containing a number of machine code subroutines residing in the expansion ROM area of the memory map.

For the new 16 & 32K PETs, this PROM will plug straight into one of the expansion ROM sockets on the main logic board. For 8K PETs, the toolkit is mounted on a small driver board which plugs into PET's expansion memory port and 2nd cassette port. After connecting and switching on, the toolkit is activated with one SYS command.

The features added by the toolkit are mainly used in the development and de-bugging of programs - hence the name. Although one or two of the routines are already available on cassette from other sources, the advantage of the toolkit is that they are constantly available and do not use up your working RAM.

The toolkit comes with extensive documentation, including a list of 'GOTCHAS' (situations in which mis-application of the command results in errors).

The following is a list of the commands, with a brief explanation and some personal review comments. One nice feature is that all these commands can be abbreviated to two letters, as per PETs BASIC commands.

AUTO

This command automatically prints the line numbers for you when keying in a program. The start line (defaults to 100 if not specified) and increment (defaults to 10) can be specified. After a line of BASIC is keyed in the next line number appears automatically.

Will save time, but not a lot.

DELETE

A real time saver - DELETE 300-500 will remove lines 300-500 inclusive. The command string can be shortened to DELETE 200-etc. as in LIST. DELETE is especially useful at the early stages in a program when different routines are being tried.

FIND

Absolutely invaluable if you wish to change or alter a program's construction. FIND will search out and print all lines containing a specified string of characters and/or words. For example, FIND GOSUB 1000 will print all the lines which contain GOSUB 1000, and FIND A1 will list all lines in which the variable A1 is used.

RENUMBER

This will renumber your program, starting from a specified line number (defaults to 100) in equal specified increments (defaults to 10). All GOTO's and GOSUB's are changed accordingly. Although a renumber facility is useful for 'opening out' a crowded program, it is not possible to renumber a small part of a program. This means that if you start your subroutines at multiples of 1000 (1000, 2000 etc.) for an easily identifiable program structure, RENUMBER will lose this - very annoying.

APPEND

Adds a previously saved program onto the end of a program in memory. Although APPEND will not interleave line numbers as will the Butterfield method, this is not a great disadvantage since APPEND is mostly used for building up programs from a library of subroutines. It does have the advantage over the Butterworth method in that is appends normal saved programs and not pseudo data files.

DUMP

Prints out the names and values of all (non-dimensioned) variables used in a program, when it has stopped. Variables are displayed in their order of creation and can be changed with screen editing. A useful feature for finding out why a program is not working as expected.

HELP

When a program halts due to an error, HELP displays the appropriate line, with the offending character highlighted in reverse field. A useful teaching aid for beginners in Syntax Errors and a time saver for advanced programmers, especially when using multiple statements per line.

TRACE, STEP, OFF

These commands are used while a program is running and display the line numbers as they are executed in a reverse field window on the screen, together with the previous five line no.'s. The speed of operation can be controlled with the SHIFT and STOP keys and STEP allows BASIC to be executed one line at a time. OFF turns TRACE and STEP off. These commands are somewhat disappointing in that they don't display the contents of the line as well as the line no.'s (unlike one

tape-based version from the States) but do at least display several line numbers at once.

The Programmers Toolkit is a Commodore Approved product. The chip for 16/32K PETs costs 35 + VAT and the version for 8K PETs - 55 + VAT.

THE PET AND A SPASTIC BOY'S POETRY

Christopher Nolan is a 14 year old spastic boy from Dublin who suffers from severe athetoid cerebral palsy. He cannot speak, and has almost no control over his movements. Three years ago teachers at a remedial school discovered that with the help of an anti-spastic drug and someone to hold his head, he had just enough control to pick out the keys on a typewriter. They found that he could not only read and spell but that he used words in a creative and chillingly apt way.

Taste of pity as people stare
Love, lots of love from mother.
Pills you find as lasting prayer
An irate person may possibly
Have faith instead of despair

Each word has been a formidable struggle against the spasms of his body. Sitting with a pointer attached to his headband, with his mother supporting his head, he directs the pointer towards the letter he wants. Sometimes his enthusiasm and frustration bring a violent spasm as he tries to select the appropriate key with his pointer.

Last December the Sunday Times magazine printed a collection of Christopher's poetry which brought acclaim from University professors, critics and writers.

Subsequently, Philip Odor, a Research Fellow, at Edinburgh University devised a PET based system to help Christopher write. Christopher sits in front of PET's screen and a moving blob of light scans rows of letters and symbols. He presses a switch with his chin to select a row then an arrow hops along the selected row and he halts it by squeezing his knees together. The letter then appears in a sentence block on the screen. If he has misfired he simply stops the blob of light at the "cancel" signal and starts again. A word processor and stored dictionary program provides Christopher with further assistance. The PET floppy disk and printer was lent to Christopher for a few weeks by Software Services Development Ltd.

In February, the "Sunday Times Magazine" launched an appeal to raise 2200 pounds to buy Christopher his PET outright. Response was overwhelming. The money needed for Christoper's system has been subscribed ten times over and will be used to provide a fund to help others with similar problems in communication. A special unit is being set up by the Central Remedial Clinic in Dublin, and Commodore are making available ten complete systems at half-price. Christopher writes of his PET:-

THE PET

My typewriter now is obsolete
It's told of my tongue-tied genetic dreams
Munificent mendicant hags
Spouted Cestus's loincloth
Cannot the awful poverty ever be halted momentarily
By a computerised friend

If you would like to help contribute towards the work of the unit please send donations to:

Christopher Nolan Appeal
Sunday Times
12 Coley Street
London WC99

SOFTWARE PRIZES
Dave Middleton

In Volume 2 Number 2, Andrew Goltz announced the introduction of the software prize for the best article in CPUCN, the prizes being 50 pounds for the best article in the issue and a 250 pound prize for the best article in the volume. Very little publicity has been given to this and I would suspect that most readers forgot all about it. The decision for who would get the prizes was fairly difficult. Jim Butterfield has been so prolific that he should by rights win all the prizes but that would be a bit pointless as he has access to any software he requires. Also a lot of material is produced 'in house', Mike Gross-Niklaus and Paul Higginbottom are prime examples.

This leaves a fairly small amount of original work produced outside Commodore UK. We have published a lot of small items about programming but there have been very few articles sent in which extend over more than one page but those we have had are generally very good.

Here are the 50 pound software prizes:

No.	Name	Article
1	D.Muir	Digital to Analogue Conversion.
2	Mike Todd	Owners Report.
4	RJ Leman	Assembling an Assembler.
5	LJ Slow	Sorting by Insertion & chaining.
7	AR Clarke CD Smith	Far infra-red astronomy ground station.

7	DA Hills	Supermon Old ROM.
8	D Doyle	DIMP.
8	R Davis	The 'ultimate' screen save.

The 250 pound software prize for volume 2 goes to:

5	Bob Sparks	TVA meter for the physics lab.

If the above people would like to write to me giving their choice of software from the current Petpack Master Library catalogue I will arrange to have the programs sent.

I hope this will tempt a few more of you into taking up the pen or better still Wordpro and putting your thoughts, applications, programs or ideas onto paper, it is suprisingly easy once you get started.

COMPUTER PHILOSOPHY

W. T. Garbutt Mississauga, Ont.

> "It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light , it was the winter of despair, we had everything before us, we had nothing before us, we were all going directly to Heaven, we were all going directly the other way--in short, the period was so much like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only."

So began Charles Dickens in the famous introduction of the 18th Century setting for "A Tale of Two Cities" over a century ago! As with all great literature, the vision Dickens gives is perhaps centuries ahead of its time, in any event just as germane today as it was when Dickens first wrote it. Today Western civilization is on a precipice; the brink of a great leap forward for Humanity or a momentary or perhaps fatal fall.

While less than one sixth of the worlds population consumes over 80% of the worlds resources, hundreds of millions fight daily to survive on the remainder. And those who survive remember!

For the fortunate few, the leap forward has begun. While scholars will argue for generations as to the precise time, (there probably is not one) the race to the Moon will likely symbolize the start for many. None of the generation of the sixties will forget President Kennedy's proclamation to reach the Moon by the end of that decade (nor will they likely forget his tragic and senseless death). The last brief decade has witnessed a massive expansion of knowledge, the extent of which is beyond human comprehension. If all knowledge known were increased by the order of a magnitude in the 70's, is there any doubt the increase will be two orders of magnitude this decade? Is there any certainty that this exponential knowledge explosion will quench humankinds' thirst by the 21st century?

Yes. There is serious doubt if the knowledge does not bring with it a maturity, a compassion for fellow man, a sharing and a sacrifice . The dismal scenario has already been summarized by Mr. Dickens in the 18th century: war, pestilence, famine and destruction. And even with our present limited knowledge is there any doubt that a similar order of a magnitude, an exponential order of horror could soon be our legacy.

There are disquieting signs that the future optimism, our traditional western belief in historical progress may be open to serious question. What Western civilization must guard against is introspection. Cultural isolation will almost certainly provide the impetus for an apocalypse. This introspection can take many forms. Most familiar to the readers of The Transactor is the discipline of electronics and more particularly computers.

Certainly this is a paradox. The computer, the single most important element in the recent knowledge explosion, the genie that could provide the key to the crucial inter-relationships in what at present is an incomprehensible mass of data, surely could not be viewed as retrenchment. On the contrary if a limited number of well educated and affluent people become preoccupied with the computer solely for their own personal intellectual gratification that is precisely what could happen.

However it need not happen. What is required is an atmosphere of freedom, freedom of thought, expression, knowledge freely accessible to all as well as freedom from hunger and poverty. A vast educational challenge exists certainly as great as that witnessed in the race to the Moon. And the micro-computer is ideally suited as a central instrument in meeting the challenge.

One of the major micro-computer applications will be communications. The micro-computer must become as universally used as the light bulb. It must be available to the 'outbacks' of civilization as well as the present temples of Western civilizations (Universities). Computer literacy must become, with literacy, the prime objective of man after the basic survival

improvements over hunger and poverty. No longer must language create communication barriers between nations, no longer may state bureaucracies hide knowledge clandestinely or inadvertently, no longer can corporations shape consumer habits through controlled ignorance.

These freedoms of course carry risks. Change creates instability. Governments and citizens fear unbridled change especially anarchy. But the risks of not accepting the challenge is greater. We could lose all the significant advances that civilization to date has witnessed. The present world population could turn away from a Western tradition that failed to solve humankinds most immediate problems.

The computer can become a central tool to manage today's knowledge, freeing man's creative and underutilized imagination, permitting him time to examine himself.

Today Western civilization senses the future. One has only to witness the long lineups and multi-billion dollar box office receipts for 'science fiction' movies. Today's audiences are seeing implemented yesterday's dreams. And super realistic depictions of tomorrow. The audiences understand, as never before, the revolutionary technological concepts. Indeed they will accept nothing less. In many instances these special effects were only made possible with the aid of the computer. We must not permit this phenomenon to become a form of escapism. It is important to feed the scientific needs of our citizens. But we must also consider the equally important spiritual needs. eg. beliefs and values.

The computer can provide assistance to man. For every self-gratifying use there is equally a use that will help meet the challenges. From simple energy conservation methods such as controlling house temperature, or arranging car pools to complex techniques to improve work effiency computer uses abound to cut waste.

A recent industry census indicated there are over 50,000 computers in use today. Within two years personal ownership is projected to quintuple. Think of the immense number of applications such a user population can generate.

As a member of the micro-computer community, ask yourself the question "What uses can I derive and implement that will aid, no matter how insignificant it may appear, in meeting the challenges of today?" Talk to fellow users , share your ideas and you will likely find yourself in the vanguard of tomorrow's leaders.

8K—16/32K COMPARISON

Collected by Jim Russo

The Operating Systems of the 16/32K PET is about 90% the same as the 8K PET, but has been re-assembled so that almost everything is in a slightly different place in memory than it used to be. Most bugs have been fixed and some limitations removed.

Any pure BASIC program (no PEEK, POKE, SYS or WAIT) that works on an 8K PET should also work on a 16/32K PET. POKing and PEEKing screen memory (32768 to 33767) is still safe but POKing the operating system (below 1024 decimal) or using an operating system PEEK value to make a decision could be hazardous. Other programs can be made to work properly if references to RAM and ROM locations are changed. Commodore's 16/32K PET manual contains a memory map for pages 0. 1 and 2. A list of new ROM addresses follows. These two lists should contain the information needed in most cases.

MAIN HARDWARE DIFFERENCES:

- The character generator ROM has been revised so that when lower case mode is selected, upper and lower case are interchanged. That is, the 'SHIFT' key must be used to obtain an upper case character. Also, 8K programs using lower case that are run on a 16/32K PET will display all lower case as upper case and vice versa.

- The signal which blanks the video on the 8K is not connected on the 16/32K, so POKE 59409,52 no longer works. The ROM routines still reference this address but the required hardware seems to have been omitted.

SUMMARY OF DIFFERENCES

- The bug in TI has been fixed. Now every 623rd interrupt doesn't increment TI. Also TI is allowed to count 1/60 sec. too far: 240000 precedes 000000.

- Execution (of at least some code) is faster due to more efficient coding and better use of zero page. PRINT (to screen) is faster because extra code to maintain separate POS pointer has been eliminated. Also, screen snow and 'scroll - up flash' has been eliminated thanks to dynamic screen RAMs.

- Standard typewriter operation i.e. shift for upper case.

- RND (0) returns a number derived from interval timers.

- OPENing more than 10 files no longer crashes system.

OPEN statement correctly sets "current tape buffer pointer".

- Machine Language Monitor included in ROM. BRK vector is initialized to monitor. 'L' and 'S' (LOAD and SAVE from monitor) have new syntax.

* - NMI vector no longer tied to +5v. NMI is initialized to BASIC 'Warm Start".

- Data file write error corrected. The Tape Output routines now wait 2/3 second after turning on motor before beginning to write tape leader. 8K PET waits 13 ms. on drive 1, 57 ms. on 2.

- Cursor home, left, right, up, down are now tracked properly by the POS function. This causes apparent differences in the TAB function which subtracts POS from its argument to determine the number of spaces needed.

- SPC(Ø) corrected.

- When output is directed to an alternative device, the ASCII space code $2Ø is used for all BASIC supplied forward spacing. 8K used $1D.

* - Screen blanking (POKE 59409,52) no longer available, however, the scroll routine still uses it as if it did. Note some control line is still used for "EoI OUT".

- PEEK is no longer restricted.

- Array dimensions now as high as 32767 (used to be 256).

* - The memory expansion port uses a different connector.

- Spaces no longer allowed in keywords (e.g. GOSUB cannot be coded as GO SUB).

- POKE and PEEK can now be used in the same line (i.e.,POKE 8ØØØ,PEEK (9ØØØ) now works).

- ST (the status word) if used, must be tested before input of file data.

- Most ROM routines and RAM addresses have changed.

	8K	16/32K
INTFLP	D278	D26D
FLPINT	DØA7	DØ9A
CHRGOT	ØØC2	ØØ7Ø
WARM START	C38B	C389
FLOATING AC	ØØBØ-ØØB6	ØØ5E-ØØ64

- BASIC input buffer is no longer in zero page so programs which used many free locations in this area must be re-written.

* - The decoding of screen memory now uses A11 (address buss line 11). Addresses 8ØØØ-8FFF (34816 to 36863 decimal) no longer address screen memory.

(* HARDWARE CHANGES)

HERE ARE SOME COMMENTS FROM MR. M.J. SMYTH who is the Senior Lecturer, Department of Astronomy, Royal Observatory, Edinburgh, EH9 3HJ.

Using BASIC and the IEEE 488 bus, PET can input 4Ø numbers per second from a 3-1/2 digit voltmeter (Hewlett Packard 3437A). Also using BASIC, the user port can generate an output trigger (e.g. to a measuring device) within about 1Ø ms of an input trigger. We have not yet tried using assembler. But the BASIC speeds make possible very interesting applications in equipment control and real-time data processing.

PET ROM GENEALOGY

From Commodore America Newsletter
Updated by:

Dave Briggs

The PET/CMB computer has been around for over two years and, in that time, various changes have been made to the system software stored in the ROMs. To bring you up to date and clarify any confusion that may have been caused by the number of different ROM sets, we are publishing the following list with explanations.

When the PET 2ØØ1 first went into production, there were two ROM sets incorporated into the system. One ROM set is the 654Ø type ROM. This is a 28 Pin ROM which is manufactured by MOS Technology Inc. You will find these ROMs in the following locations on the PET 2ØØ1-4K and 2ØØ1-8K Main Logic Board:

Location	ROM	Part Number
H1	654Ø-Ø19	9Ø1439-Ø9 or Ø1
H2	654Ø-Ø13	9Ø1439-Ø2
H3	654Ø-Ø15	9Ø1439-Ø3
H4	654Ø-Ø16	9Ø1439-Ø4
H5	654Ø-Ø12	9Ø1439-Ø5
H6	654Ø-Ø14	9Ø1439-Ø6
H7	654Ø-Ø18	9Ø1439-Ø7
A2	654Ø-Ø1Ø	9Ø1439-Ø8

NOTE: There is an Ø19 ROM at the H1 location. On some earlier Main Logic Boards you will find a 654Ø-Ø11 at H1.

This ROM has been updated to an 019 due to an intermittent bug in the edit software. This ROM Set is Basic level 1.

The other ROM Set incorporated into the PET 2001 is a type 2316B 24 Pin ROM. You will find these ROMs in the following locations on the PET 2001-4K and 2001-8K Main Logic Board:

Location	ROM	Part Number
H1	901447-09	901447-09
H2	901447-03	901447-03
H3	901447-05	901447-05
H4	901447-06	901447-06
H5	901447-02	901447-02
H6	901447-04	901447-04
H7	901447-07	901447-07
A2	901447-08	901447-08

NOTE: There is an 09 ROM at the H1 location. On some earlier Main Logic Boards you will find a 901447-01 ROM. This ROM has been updated to an 09 ROM due to an intermittent bug in the edit software. Like the 6540 ROM Set, this too is a Basic Level 1 ROM Set. To determine what the 6540 and 2316B ROMs listed above are capable of, I would refer you to the "PET User Manual", Model 2001-8.

The next two ROM Sets are Basic Level II ROMs, and are fitted as standard on all 16 and 32K PET/CBMs. They are also Retrofit Kits for the 2316B and 6540 Basic Level 1 ROMs. The Basic Level II ROMs include the machine language. Basic Level II allows you to interface the Commodore 2040 Dual Floppy to your PET/CBM. Basic Level I ROMs will not allow you to interface the 2040 Dual Floppy to your PET. The Basic Level II Retrofit ROMs also allows you to use arrays with more than 255 elements.

If your PET/CBM has the Basic Level I 6540 ROMs, you could use the following ROMs which come in the form of a Retrofit Kit to upgrade your PET/CBM to Basic Level II:

Location	ROM	Part Number
H1	6540-020	901439-13
H2	6540-022	901439-15
H3	6540-024	901439-17
H4	6540-025	901439-18
H5	6540-021	901439-14
H6	6540-023	901439-16
H7	6540-026	901439-19

If your PET/CBM has the Basic Level I 2316B ROMs, you would use the following ROMs which come in the form of a Retrofit Kit to upgrade your PET/CBM to Basic Level II:

Location	ROM	Part Number
H1	901465-01	901465-01
H2	901465-02	901465-02
H3	901465-24	901465-24
H4	901465-03	901465-03
H5	Blank	
H6	Blank	
H7	Blank	

The following ROM Sets are the ROMs that are currently being used in production. There are two sets of ROMs in use. If you have a graphic style PET, you should have the following ROMs in your unit:

Location	ROM	Part Number
D3	Blank	
D4	Blank	
D5	Blank	
D6	901465-01	901465-01
D7	901465-02	901465-02
D8	901447-24	901447-24
D9	901465-03	901465-03
F10	901447-10	901447-10

If your computer is a business style, you should have the following ROMs in your unit:

Location	ROM	Part Number
D3	Blank	
D4	Blank	
D5	Blank	
D6	901465-01	901465-01
D7	901465-02	901465-02
D8	901447-01	901447-01
D9	901465-03	901465-03

The ROMs in the graphic and business PET/CBM are Basic Level III ROMs. To determine what any machine fitted with Basic Level III is capable of, you should refer to the "CBM User Manual" Model 2001-16, 16N, 32, 32N.

The ROMs currently being used in production of the 3040 Dual Floppy are as follows:

Location	ROM	Part Number
UL1	901468-06	901465-06
UK1	Blank	
UH1	901465-07	901468-07
UK3	6530-02	901466-02

These ROMs are DOS Version I.

New 80 column PET BASIC 4.0 DOS 2.1 Retrofit ROMs will be available in about 60 days time at 38.00 BASIC 4.0, 38.00 DOS 2.1.

New BASIC eliminates garbage collection problems and has a powerful direct access feature which considerably enhances the disk system.

BASIC has been extended to include a number of D.U.M. (Disk Utility Maintenance) features, considerably improving "User Friendliness". DOS 2.1 simply enables the new commands to be recognised.

SPACE INVADERS

I, for one, have worn the 'A' key out on my PET. Here is an ideal solution by G. Luxford.....
"I have found the SPACE INVADER program

to be a knock-out, but the key for beam firing, "A" is coming in for severe pounding, particularly when other over enthusiatic hands get onto the keyboard.

With the new ROM version a temporary solution is to shift the commands onto temporary keys. This can be achieved by POKE1409,6 after loading but prior to running the program, to shift commands to Z, 1 and 3. This locations can alternatively be POKEd with any value from 0 to 9, except 4 (present values) to select other options for the keyboard. Another option is to POKE1938 with 2, 4, 8 or 16 to shift the beam fixing key along the key row.

A better solution is to shift the control to the user port input. This can be achieved by POKE1414,79 and POKE1417,79. Control is then by switch closers to the ground line, pin N of PA0 on pin C to fire beam, PA7 on pin L to move the laser base right and PA6 on pin K to move the laser base left. You can now use large hefty buttons or a joy stick control with complete immunity to keyboard damage by over-excited saucer shooters.

You may SAVE the program either by BASIC without a name or by TIM monitor, from $0400 to $2000.

The program is also being modified to give the option of any alpha key to fire the beam, any numeric key to shift the laser beam and the option of control by the user port. This requires appreciably more patching and is not yet fully debugged.

When completed I will send details with a revised program to CPUCN.

I regret not being of direct help to old ROM users, having recently become a new ROMer, but can only suggest they try the recommended program mods and see what happens. There seems a good chance this will work.

A REVIEW OF ADVENTURE
Dave Middleton

Adventure is a very difficult game to describe because it is so unusual and because if too much information is given away then the game is spoilt.

First a very short history of the game. It was written by two programmers Willie Crowther and Don Woods at the Massachusetts Institute of Technology as an exercise in Artificial Intelligence. The original program was written in FORTRAN which is quite amazing as FORTRAN is not exactly renowned for its string handling capabilities! The game has had an enormous secret following for years as just about every main-frame computer has a copy lurking in the depths of its disk packs which is played by a small band of devotees at enormous cost to the company.

It can hardly be a suprise that PET Adventure has been made available by Jim Butterfield who has converted the FORTRAN code into its BASIC equivalent. The main Adventure program is 12k long and when the text files are included this comes to a massive 56k! To run Adventure you will need a 32k PET and a disk system with Adventure running on drive 0.

The program is text oriented which in this case is far better than trying to use graphics as a few well chosen words can build an imaginative picture that would take Walt Disney a month to put on film. I cannot describe any of the events which occur but maybe the following will give you the flavour of the game....

'Somewhere nearby is a colossal cave, where others have found fortunes in treasure and gold, though it is rumoured that some who enter are never seen again. Magic is said to work in the caves. I will be your eyes and hands....'

The game is organised around a complex cave system with various challenges and objects hidden away within it. As the caves are explored and the treasure removed points are given with a maximum score of 300 being possible. I have never met anybody who has achieved this by themselves without either looking at the source code or taking clues from friends. How do I rate as an Adventurer? So far I have only scored 64 points!

If any User Club would like a copy of Adventure then please contact me at Commodore Slough with details of your club activities and I will send it to you. If there is no group in your area and you are willing to set one up then please contact me. Please note that I am unable to send out copies of Adventure to individual members at this time.

USER GROUPS

The following people are contacts for those of you who may be interested in joining IPUG (Independent PET User Group). Some of the groups are well established and others have only been formed recently, I would suggest that if you wish to meet other PET users for a chat or maybe a bit of assistance with programming problems you contact either the nearest group to you or Eli Pamphlett at the address given below for more details.

General Secretary:
Eli Pamphlett
The Coppers
Sudbury Road
Yoxall
Burton Upon Trent
Staffs
Tel. Yoxall (0543) 472222

Mrs J.I Rich
Central Electricity Gen. Board
Research and Development Dept.
Berkley Nuclear Lab.
Berkley
Glos.

G.A Parkin
Robert May's School
West Street
Odiham
Hants

Maurice D Meredith
Lect. Comp & Info Studies
Dept of Education
The University
Southampton

Wg. Cdr. M.A.F Ryan
164 Chesterfield Drive
Riverhead
Sevenoaks
Kent

Mr Franklin
Petalect Ltd.
33/35 Portugal Road
Woking
Surrey

Paul Handover
32 Long Wyre Street
Colchester
Essex

Brian Broomfield
Little Orchard
Hill Farm
Radlett
Herts

F.J Townsend
The Mill
Rhydowen
Llandyssul
Dyfed

M.J Merriman
"Pippins"
The Ridgeway
Stourport on Severn

Raymond N. Davies
105 Normanton Rd
Derby

David W Jowett
197 Victoria Road East
Thorton-Cleveleys
Blackpool

Eliot Khabie-Zeitoune
9 Hamlea Close
London SE12

Geoff Squibb
108 Teddington Pk Rd
Teddington
Middex

Jim Cocallis
20 Wrocester Rd
Newton Hall Estate
Durham

Trevor Langford
Allen Computers
16 Hainton Avenue
Grimsby
South Humberside

Dr J MacBrayne
27 Paidmyre Cresent
Newton Mearns
Glasgow

Geoff is interested in setting up a user group in the Slough area, if anybody is interested then please contact him at the address below.

Geoff Luxford
51 Station Rd
Burnham
Nr Slough

Tel Burnham 2601

The following groups are not affiliated to IPUG but have some of the largest group memberships outside the Official Commodore User Club.

North London Computer Club
North London Poly.
Holloway
London N7

S.U.P.A
(Southern Users of Pet Association)
Mr H. Pilgrim
143e Ditchling Rd
Brigton

Kent and Sussex User Group

A. French
Crawley Teacher Centre
Ashdown Drive
Crawley

or

A. Upnown
West Sussex Institute
Bognor College
Bognor Regis
Sussex

If I have omitted any user group from the above list then I apologise in advance but the problem is lack of communication. To this end I am going to keep a page of CPUCN set aside for club news so that the independent clubs can announce meetings and events.

Memory Maps

BASIC 1 MEMORY MAP

```
000             $4C constant (6502 JMP instruction)
001-002         USR function address lo, hi

Terminal I/O maintenance

003             Active I/O channel #
004             Nulls to print for CRLF (unused).
005             Column Basic is printing next
006             Terminal width (unused).
007             Limit for scanning source colmns (unused)
008             Line number before storage buffer. (integer address from Basic)
009             $2C constant (special comma for INPUT process).
010-089         BASIC INPUT buffer (80 bytes).
090             General counter for BASIC. (search char ':' or endline)
091             $00 used as delimeter (scan between quotes flag).
092             General counter for BASIC. input buffer pointer.
093             Flag to remember dimensioned variables. 1st char of name .
094             Flag for variable type: 0=numeric; 1=string.
095             Flag for integer type: 80=integer; 00=floating point.
096             Flag to crunch reserved words (protects "& remark).
097             Flag which allows subscripts in syntax.
098             Flags INPUT or READ: 0=Input; 64=Get; 152=Read.
099             Flag sign of TAN.
100             Flag to suppress OUTPUT (+normal;-suppressed).
101             Index to next available descriptor.
102-103         Pointer to last string temporary lo; hi.
104-111         Table of double byte descriptors which point to variables.
112-113         Indirect index #1 lo; hi.
114-115         Indirect index #2 lo; hi.
116-121         Pseudo register for function operands.

Data storage maintenance

122-123         Pointer to start of BASIC text area lo; hi type
124-125         Pointer to start of variables lo; hi byte.
```

126-127	Pointer to array table lo; hi byte.
128-129	Pointer to end of variables lo; hi byte.
130-131	Pointer to start of strings lo; hi byte.
132-133	Pointer to top of string space lo; hi byte.
134-135	Highest RAM adr lo;hi byte.
136-137	Current line being executed. A zero in 136 means statment executed in a direct command.
138-139	Line # for continue command lo; hi.
140-141	Pointer to next STMNT to execute lo; hi.
142-143	Data line # for errors lo; hi.
144-145	Data statment pointer lo; hi.(145-memory address of data line)

Expression evaluation

146-147	Source of INPUT lo; hi.
148-149	Current variable name.
150-151	Pointer to variable in memory lo; hi.
152-153	Pointer to variable referred to in current FOR-NEXT
154-155	Pointer to current operator in table lo; hi.
156	Special mask for current operator.
157-158	Pointer for function definition lo; hi.
159-160	Pointer to a string descriptor lo; hi.
161	Length of a string of above string.
162	Constant used by garbage collect routine.(3or7 for grbg clct)
163	$4C constant (6502 JMP inst).
164-165	Vector for function dispatch lo; hi.
166-171	Floating accumulator # 3
172-173	Block transfer pointer # 1 lo;hi.
174-175	Block transfer pointer # 2 lo; hi.
176-181	Floating accumulator # 1(FAC#1)(USR function evaluated here).
182	Duplicate copy of sign of mantissa of FAC # 1.
183	Counter for # of bits to shaft to normalize FAC # 1.
184-189	Floating accumulator # 2.(FAC#2)
190	Overflow byte for floating argument.
191	Duplicate copy of sign of mantissa.
192-193	Pointer to ASCII rep of FAC in conversion routine lo; hi.

RAM subroutines

194-199	CHARGOT RAM code. Gets next character from BASIC text.
200	CHARGOT RAM code regets current characters.
201-202	Pointer to source text lo; hi.

203-223 Next random number in storage

OS page zero storage

224-225 Pointer to start of line cursor loc lo; hi.
226 Column positionof cursor.(0-79).
227-228 General purpose start address indirect lo; hi.
229-233 General purpose end address direct lo; hi.
234 Flag for quote mode on/off.
235 timer 1 interrupt sttus: 0=disabled
236 EOT character received
237 character error received
238 current file name length.
239 Current logical file number.
240 Current primary address.
241-242 Current secondary address.(241 device no; 242 max line length)
243-244 Pointer to start of current tape buffer lo; hi.
245 Current screen line #.
246 Data temporary for I/O.
247-248 Pointer to start loc for O.S. lo-hi.(tape start adress/pointer)
249-250 Pointer to current file name lo; hi.
251-254 Tape variable storage.
255 Overflow byte BASIC uses when doing FAC to ACIII conversions.

Page 1

62 bytes on bottom are used for error correction in tape reads. Also, buffer for ASCII when Basic is expanding the FAC into a printable number. The rest of page 1 is used for storage of BASIC GOSUB and FOR NEXT context and hardware stack for the machine.

Page 2

512-514 24-hour clock in 1/60 secs
515 Matrix co-ordinates of last key down (row/col; 255=no key)
516 Shift key status: 0=no shift; 1=shift
517-518 Correction factor for clock, LSB, MSB
519-520 Interrupt driver flag for cassette # 1,switches; # 2 switches (519 for cassette#1 on; 520 for cassette#2 on)
521 Keyswitch PIA duplicate of 59910

522	timing constant buffer
523	Flag # means verify not load into memory.
524	I/O status byte.
525	Index into keystroke buffer.
526	Flag to indicate reverse-field on.
527-536	Interrupt driven key stroke buffer.
537-538	IRQ RAM VECTOR lo; hi.
539-540	BRK instruction RAM VECTOR lo; hi.
541	(IEEE mode)
542	(end of line for input pointer; # characters on screen line)
543	?
544-545	(cursor log row/col, used in input routines)
546	(PBD image for tape I/O)
547	Keyboard input code.
548	Blink cursor flag.
549	Count down to flip cursor. Cursor blink duration.
550	Screen value of input character when cursor moves on.
551	Flag for cursor on/off.
552	(EOT bit received,tape write)
553-577	Table of LSB of start address of video display lines (25).
578-587	Table of logical addresses.
588-597	Table of primary addresses.
598-609	Table of secondary addresses.
608	Input from screen/keyboard flag. 0=keyboard; 1=screen.
610	Index into LA,FA,SA, tables
611	Default input device #.
612	Default output device #.
613	Computation of parity on cassette write.
615-615	?
616	Tape buffer item counter.
617-619	?
620	Serial bit count.
621	Count of redundant tape blocks.
622	?
623	(cycle counter,flip for every bit coming from tape)
624	Count down synchronization or cassette write.
625-626	Index next character in/out tape buffer # 1; # 2.
627	Countdown synchronization on cassette header.
628	Flag to indicate bit/byte error.
629	Flag to indicate tape routine reading shorts.

630-631	Index to addresses to correct on tape read pass 1; pass.
632	Flag for cassette read-tells current function-countdown,read,etc
633	Count of seconds of shorts to write before data.
634-825	Buffer for cassette # 1 (192 bytes)
836-1017	Buffer for cassette # 2 (192 bytes)
1018-1023	Unused.

SUBROUTINE LOCATIONS IN BASIC 1 MACHINES

C000-C091	keyword action addresses
C092-C18F	table of reserved words
C190-C2AB	error messages
C2AC-C2D9	peeks at the stack for active FOR loop
C2AD-C31C	'open up' a space in Basic for insertion of a new line
C31D-C329	tests for stack-too-deep and aborts if found.
C32A-C356	check available memory space
C357-C388	sends a canned error message from C190 area, then drops into:
C389-C391	signals 'ready' (C38B entry for basic warm start).
C394-C3A9	gets a line of input, analyses it, executes it
C3AC-C42E	handles a new line of Basic from keyboard; deletes old line etc.
C430-C460	corrects the chaining between Basic lines after insert/delete
C462-C476	receives a line from the keyboard into the Basic buffer
C479-C48C	gets each character from keyboard
C48D-C521	looks up the keywords in an input line and changes to "tokens"
C522-C550	searches for the location of a Basic line from number in 8,9
C551-C599	implements NEW command - clears everything
C59A-C5A7	sets the Basic pointer to start-of-program
C5A8-C647	performs LIST command
C649-C68F	executes a FOR statement
C692-C6B4	continues to build FOR vectors
C6B5-C6EF	reads and executes the next Basic statement, find next line, etc.
C6F2-C70A	executes the Basic Command as a subroutine
C70D-C71B	performs RESTORE
C71C-C742	handles STOP, END, and BREAK procedures
C745-C75E	performs CONT
C75F-C76D	set pause after carriage return (never called)
C770-C772	performs CLR

```
C775-C77D  performs RUN
C780-C79A  performs GOSUB
C79D-C7C9  performs GOTO
C7CA-C7FD  performs RETURN
C7FE-C81E  scans for start of next Basic line
C820-C840  performs IF
C843-C862  performs ON
C863-C89A  gets a fixed point number and stores in 8,9
C89D-C91B  performs LET
C91C-C97E  check numeric digit/move string pointer
C97F-C982  performs PRINT#
C985-C996  performs CMD
C999-CA24  performs PRINT
CA27-CA41  print string from address in Y,A
CA44-CA76  print a character
CA77-CA9E  handles bad input data
CA9F-CAC5  performs GET
CAC6-CADF  performs INPUT#
CAE0-CB14  performs INPUT
CB17-CB21  prompts and receives the input
CB24-CC11  performs READ
CC12-CC35  canned messages: EXTRA IGNORED;REDO FROM START
CC36-CC8F  performs NEXT
CC92-CCB5  checks Basic format,data type, flags TYPE MISMATCH
CCB8-CD38  inputs and evaluates any expression (numeric or string)
CD3A-CD9C  pushes a partialy evaluated argument to the stack
CD9C-CDB9  evaluates a numeric variable, pi, or identifies other symbols
CDBC-CDC0  value of pi in floating binary
CDC1-CDE7  checks for special characters at start of expression
CDE8-CDF6  performs NOT function
CDF7-CE04  performs various functions
CE05-CE0C  evaluates expression within parentheses()
CE0B-CE0D  checks for right parentheses )
CE0E-CE10  checks for left parentheses  (
CE11-CE1B  checks for comma
CE1C-CE20  prints SYNTAX ERROR and exits
CE21-CE27  sets up function for future evaluation
CE28-CE39  set up a variable name search
CE3B-CE96  check for special variables,TI,TI$,and ST
CE97-CED5  identifies and sets up function references
```

CED6-CF05 performs the OR and AND function
CF06-CF6D performs comparisons
CF6E-CF7A sets up DIM execution
CF7B-D00E searches for a Basic variable
D00F-D078 creates a new Basic variable
D079-D087 logs Basic variable location
D088-D098 array pointer subroutine
D099-D09C is 32768 in floating binary
D09D-D0B8 floating point to fixed point conversion for singal values
D0B9-D263 locates and/or creates arrays
D264-D277 performs FRE function
D278-D284 converts fixed point to floating
D285-D28A performs POS function
D28B-D294 checks direct/indirect command, gives 'ILLEGAL DIRECT'
D295-D348 executes DEF statements and evaluates FN(X)
D349-D36A performs STR$ function
D36B-D3D1 scans and sets up string elements
D3D2-D403 builds string vectors
D404-D5C3 does 'garbage collection' -discards unwanted strings
D5C4-D5D7 performs CHR$ function
D5D8-D653 performs LEFT$, RIGHT$, MID$, functions
D654-D662 performs LEN, gets string length
D663-D672 performs ASC function
D673-D684 gets a single byte value from Basic
D685-D6C3 evaluates VAL function
D6C4-D6CF gets two arguments (16 bit and 8 bit) from Basic
D6E6-D701 performs PEEK and POKE
D702-D71D executes WAIT statement
D71E-D890 performs addition and subtraction
D891-D8BE contains floating-point constants
D8BF-D8FC performs LOG function
D8FD-D95D performs multiplication
D95E-D988 loads secondary acumulator from memory ($B8 to $BD)
D989-D9B3 test and adjust primary/secondary accumulators
D9B4-D9E0 routines to multiply or divide by 10
D9E1-DA73 performs division
DA74-DA98 loads primary accumulator from memory ($b0-$B5)
DA99-DACD transfers primary accumulator to memory
DACE-DADD transfers secondary accumulator to primary
DADE-DAEC transfers primary accumulator to secondary

DAED-DAFC	rounds the primary accumulator
DAFD-DB29	extracts primary sign; performs SGN function
DB2A-DB2C	performs ABS
DB2D-DB6C	compares primary accumulator to memory
DB6D-DB9D	Convert Floating point to fixed, unsigned
DB9E-DBC4	perform INT function
DBC5-DC4F	convert ASCII string to floating point
DC50-DC84	get new ASCII digit
DC94-DCAE	print Basic Line number
DCAF-DDE2	convert floating point to ASCII string (at 0100 up)
DDE3-DE23	conversion constants - decimal or clock
DE24-DE2D	evaluation SQR function
DE2E-DE66	evaluation of power function
DE67-DE71	negate (monadic -)
DEA0-DEF2	perform EXP function
DEF3-DF3C	perform function series evaluation
DF45-DF9D	perform RND calculation
DF9E-DFA4	evaluate COS function
DFA5-DFED	evaluate SIN function
DFEE-E019	evaluate TAN function
E048-E077	evaluate ATN function
E0B5-E0CC	Basic scan program, transferred to 00C2-00D9
E0D2-E172	completion of power-on-reset; memory test, etc.
E19B-E1BB	partial test for TI and TI$
E1BC-E1E0	input/read/get director
E1E1-E27C	initialize I/O registers, clear screen, reset subroutines
E27D-E3C3	receive input f om keyboard/screen
E3C4-E3E9	set up new screen line
E3EA-E52F	output character to screen
E530-E5DA	check or and perform screen scrolling
E5DB-E66A	start new screen line
E66B-E67D	interrupt entry
E67E-E683	interrupt return
E685-E73E	hardware interrupt routine: cursor flash, tape monitor, keyboard
E73F-E7AB	convert keyboard matrix to ASCII
E7AC-E7B9	write-on-screen subroutine
E7DE-E7EB	print canned monitor message
F0B6-F1CB	IEEE-488 channel open, test, close
F1CC-F22F	get input character from keyboard, screen cassette, IEEE
F230-F27C	output character to screen, cassette, IEEE

F27D-F2A3 restore normal I/O, clear IEEE channels
F2A4-F2AA abort (not close!) all files
F2AB-F2B7 locate logical file table entry
F2B8-F2C7 transfer file table entries to Device, Command
F2C8-F329 perform file CLOSE
F32A-F33E test stop key
F33F-F345 test if direct/indirect command for suppressing file advice
F346-F3FE perform file LOAD
F3FF-F421 print "SEARCHING.."
F422-F432 print "LOADING.." or "VERIFYING"
F433-F461 get parameters for LOAD and SAVE
F462-F494 perform IEEE sequences for LOAD, SAVE, and OPEN
F495-F4BA search for specific tape header
F4BB-F4D3 perform VERIFY
F4D4-F529 get parameters for OPEN and CLOSE
F52A-F5AD perform OPEN
F5AE-F5E2 search for any tape header
F5E3-F5EC clear tape buffer
F5ED-F64C write tape header
F64D-F666 get start and end addresses from tape header
F667-F67C set buffer start address
F67D-F694 set tape buffer start and end pointers
F695-F69D perform SYS command
F69E-F71B perform SAVE
F71C-F735 find unused secondary address
F736-F78A update clock
F78B-F7DB set input device
F7DC-F82C set output device
F82D-F83A bump tape buffer counter
F83B-F85D wait for cassette PLAY switch
F85E-E870 test cassette switch line
F871-F87E wait for cassette RECORD and PLAY switches
F87F-F8B8 read tape initiation routine
F8B9-F8D1 write tape initiation routine
F8D2-F912 complete tape read or write
F913-F91D wait for I/O completion
F91E-F92D test stop key and abort if necessary
F92E-F95E subroutine to set tape read timing
F95F-FBFB interrupt routine for tape read
FBDC-FBE4 save memory pointer

```
FBE5-FBEB  set ST error flag
FBEC-FBFF  subroutine to count 8 serial bits per byte
FC00-FC1B  subroutine to write a bit to tape
FC1C-FCFA  interrupt 1 for tap write - entry at FC21
FCFB-FD15  terminate I/O and restore normal vectors
FD16-FD37  subroutine to set interrupt vector
FD38-FD47  power-on reset entry; test for diagnostic
FD48-FD7B  diagnostic routine
FD7C-FD8F  checksum routine
FD90-FD9A  pointer advance subroutine
FD9B-FFB1  diagnostic routines
JUMP TABLE:
FFC0       OPEN
FFC3       CLOSE
FFC6       set input device
FFC9       set output device
FFCC       restore normal I/O devices
FFCF       input character (from screen)
FFD2       output character
FFD5       LOAD
FFD8       SAVE
FFDB       VERIFY
FFDE       SYS
FFE1       test stop key
FFE4       get character from keyboard buffer
FFE7       abort all I/O channels
FFEA       update clock
FFED-FFFA  turn off cassette motors
FFFA-FFFB  NMI vector (mangled)
FFFC-FFFD  reset vector
FFFE-FFFF  interrupt vector
```

BASIC 2 MEMORY MAP

0000-0002	0-2	USR Jump instruction lo-hi
0003	3	General counter for Basic. Search character ':' or endline
0004	4	Scan-between-quotes flag. 00 as delimeter
0005	5	Basic input buffer pointer; # subscripts
0006	6	Default DIM flag. First character of array name
0007	7	Variable flag, type: FF=string, 00=numeric
0008	8	Integer flag, type: 80=integer, 00=floating point
0009	9	DATA scan flag; LIST quote flag; memory flag
000A	10	Subscript flag; FNx flag
000B	11	Flags for input or read, 0==input: 64=get: 152=read
000C	12	ATN sign flag: comparison evaluation flag
000D	13	input flag; suppress output if negative
000E	14	current I/O device for prompt-suppress
0011-0012	17-18	Basic integer address (for SYS, GOTO etc)
0013	19	Temporary string descriptor stack pointer
0014-0015	20-21	Last temporary string vector
0016-001E	22-30	Stack of descriptors for temporary strings
001F-0020	31-32	Pointer for number transfer
0021-0022	33-34	Misc.number pointer
0023-0027	35-39	product staging area for multiplication
0028-0029	40-41	Pointer: Start-of-Basic memory
002A-002B	42-43	Pointer: End-of-Basic, Start-of-Variables
002C-002D	44-45	Pointer:End-of-Variables,Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: Bottom-of-Strings (moving down)
0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit of Basic Memory
0036-0037	54-55	Current Basic line number
0038-0039	56-57	Previous Basic line number
003A-003B	58-59	Pointer to Basic statement (for CONT)
003C-003D	60-61	Line number, current DATA line
003E-003F	62-63	Pointer to current DATA item
0040-0041	64-65	Input vector
0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/Next
0048-0049	72-73	Y save register-new operator save; current operator pointer

004A	74	Special mask for current operator; comparison symbol
004B-004C	75-76	Misc numeric work area; function definition pointer,lo-hi
004D-004E	77-78	Work area; pointer to string description
004F	79	Length of above string
0050	80	constant used by garbage collect routine, 3 or 7
0051-0053	81-83	Jump vector for functions
0054-0058	84-88	Misc numeric storage area
0059-005D	89-93	Misc numeric storage area
005E-0063	94-99	Accumulator#1: E,M,M,M,M,S
0064	100	Series evaluation constant pointer
0065	101	Accumulator hi-order propogation word
0066-006B	102-107	Accumulator#2
006C	108	Sign comparison, primary vs. secondary
006D	109	Low-order rounding byte for Acc#1
006E-006F	110-111	Cassette buffer length/Series Pointer
0070-0087	112-135	Subrtn: Get Basic Char; 77,78=pointer
0088-008C	136-140	RND storage and work area
008D-008F	141-143	Jiffy clock for TI and TI$
0090-0091	144-145	IRQ RAM vector,lo-hi; hardware interrupt vector
0092-0093	146-147	Break interrupt vector
0094-0095	148-149	NMI RAM interrupt vector,lo-hi
0096	150	Status word ST
0097	151	Which key depressed: 255=no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Clock correction factor;lsb-msb; 1/30 sec increment
009B	155	Keyswitch PIA duplicate of 59410 : STOP and RVS flags
009C	156	Timing constant buffer
009D	157	Load=0, Verify=1
009E	158	# characters in keyboard buffer
009F	159	Screen reverse flag
00A0	160	IEEE-488 output flag: FF=character waiting
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row,column)
00A5	165	IEEE-488 output character buffer
00A6	166	Key image
00A7	167	0=flashing cursor, else no cursor
00A8	168	Countdown for cursor timing
00A9	169	Character under cursor
00AA	170	Cursor blink flag
00AB	171	EOT bit received

00AC	172	Input from screen/input from keyboard
00AD	173	X save flag
00AE	174	How many open files; pointer into file table
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally default of 3
00B1	177	Tape character parity
00B2	178	Byte reveived flag
00B4	180	Tape buffer character
00B5	181	Pointer in filename transfer
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Countdown for tape write; sync on tape header
00BB	187	Tape buffer#1 count
00BC	188	Tape buffer#2 count
00BD	189	Write leader count; Read pass1/pass2
00BE	190	Write new byte; Read error flag
00BF	191	Write start bit; Read bit seq error
00C0	192	Pass 1 error log pointer
00C1	193	Pass 2 error correction pointer
00C2	194	Current function; 0-Scan; 1-15=Count; $40=Load; $80=End
00C3	195	Read checksum; Write leader length
00C4-00C5	196-197	Pointer to screen line
00C6	198	Column position of cursor on above line (0-79)
00C7-00C8	199-200	Utility pointer: tape buffer,scrolling
00C9-00CA	201-202	Tape end address/end of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	Flag for quote mode 0=direct cursor, else programmed cursor
00CE	206	Timer 1 enabled for tape read; 00=disabled
00CF	207	EOT signal received from tape
00D0	208	Read character error
00D1	209	# characters in file name
00D2	210	Current logical file number
00D3	211	Current secondary addrs, or R/W command
00D4	212	Current device number
00D5	213	Line length (40 or 80) for screen
00D6-00D7	214-215	Start of tape buffer, address
00D8	216	Line where cursor lives
00D9	217	Last key input; buffer checksum; bit buffer
00DA-00DB	218-219	Pointer to current file name
00DC	220	Number of keyboard INSERTs outstanding

00DD	221	Write shift word/Receive input character
00DE	222	#blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	Screen line table: hi order address & line wrap
00F9	249	Interrupt driver flag for cassette#1 status switch
00FA	250	Interrupt driver flag for cassette#2 status switch
00FB-00FC	251-252	Tape start address
0100-010A	256-266	Binary to ASII conversion area
0100-013E	256-318	Tape read error log for correction
0100-01FF	256-511	Processor stack area
0200-0250	512-592	Basic input buffer
0200-0201	512-513	Program counter
0202	514	is processor status
0203	515	is accumulator
0204	516	X index
0205	517	Y index
0206	518	stack pointer
0207-0208	519-520	user modifiable IRQ
0251-025A	593-602	Logical file number table
025B-0264	603-612	Device number table
0265-026E	613-622	Secondary address, or R/W cmd, table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape#1 buffer
033A-03F9	826-1017	Tape#2 buffer
03FA-03FB	1018-1019	Vector for Machine Language Monitor
0400-7FFF	1024-32767	Available RAM including expansion
8000-8FFF	32768-36863	Video RAM
9000-BFFF	36864-49151	Available ROM expansion area
C000-E0F8	49152-57592	Microsoft Basic interpreter
E0F9-E7FF	57593-59391	Keyboard, screen, interrupt programs
E810-E813	59408-59411	PIA 1 - Keyboard I/O
E820-E823	59424-59427	PIA 2 - IEEE-488 I/O
E840-E84F	59456-59471	VIA - I/O and timers
F000-FFFF	61440-65535	Reset, tape, diagnostics, monitor

SUBROUTINE LOCATIONS IN BASIC 2 MACHINES

C000-C045	Action addresses for primary keywords
C046-C073	Action addresses for functions
C074-C091	Hierarchy and action addresses for operators
C092-C192	Table of Basic keywords
C193-C2A9	Basic messages, mostly error messages
C2AA-C2D7	Search stack for FOR or GOSUB activity
C2D8-C31A	Open up space in memory
C31B-C327	Test: stack too deep?
C328-C354	Check available memory
C355-C388	Send canned error message, then:
C389-C3AA	Print Ready.
C3AB-C441	Handle new Basic line from keyboard
C442-C46E	Rebuild chaining of Basic lines in memory
C46F-C494	Receive line from keyboard
C495-C52B	Change keywords to Basic tokens
C52C-C55A	Search Basic for a given Basic line number
C55B-C576	Perform NEW, then:
C577-C5A6	Perform CLR
C5A7-C5B4	Reset Basic execution to start-of-program
C5B5-C657	Perform LIST
C658-C6FF	Perform FOR
C700-C72F	Execute Basic statment
C730-C73E	Perform Restore
C73F-C76A	Perform STOP and END
C76B-C784	Perform CONT
C785-C78F	Perform RUN
C790-C7AC	Perform GOSUB
C7AD-C7D9	Perform GOTO
C7DA-C7F2	Perform RETURN, and perhaps:
C7F3-C80D	Perform DATA, i.e., skip rest of statment
C80E-C810	Scan for next Basic statment
C811-C82F	Scan for next Basic line
C830-C842	Perform IF, and perhaps:
C843-C852	Perform REM, i.e., skip rest of line
C853-C872	Perform ON
C873-C8AC	Get fixed-point number from Basic
C8AD-C927	Perform LET

```
C928-C936  Add ASCII digit to accumulator #1
C937-C98A  Continue to perform LET
C98B-C990  Perform PRINT#
C991-C9A4  Perform CMD
C9A5-CA1B  Perform Print
CA1C-CA38  Print string from memory
CA39-CA4E  Print single format character ( space, cursor-right,?)
CA4F-CA7C  Handle bad input data
CA7D-CAA6  Perform GET
CAA7-CAC0  Perform INPUT#
CAC1-CAF9  Perform INPUT
CAFA-CB06  Prompt  and receive input
CB07-CBFB  Perform READ; common routines used by INPUT and GET
CBFC-CC1F  Messages: EXTRA IGNORED, REDO FROM START
CC20-CC78  Perform NEXT
CC79-CC9E  Check data type, print TYPE MISMATCH
CC9F-CDEB  Input & evaluate any expression (numeric or string)
CDEC-CDF1  Evaluate expression within parentheses ()
CDF2-CDF4  Check right parenthesis )
CDF5-CDF7  Check left parenthesis (
CDF8-CE02  Check for comma
CE03-CE07  Print SYNTAX ERROR and exit
CE08-CE0E  Set up function for future evaluation
CE0F-CE88  Search for variable name
CE89-CEC7  Identify and set up function references
CEC8-CECA  Perform OR
CECB-CEF7  Perform AND
CEF8-CF5F  Perform comparisons,string or numeric
CF60-CF6C  Perform DIM
CF6D-CFF6  Search for variable location in memory
CFF7-D000  Check if ASCII character is alphabetic
D001-D077  Create new Basic variable
D078-D088  Array pointer subroutine
D089-D08C  32768 in floating binary
D08D-D0AB  Evaluate expression for positive integer
D0AC-D227  Find or create array
D228-D258  Compute array subscript size
D259       Perform FRE
D26D-D279  Convert fixed point to floating point
D27A-D27F  Perform POS
```

```
D280-D28C  Check if direct command, print ILLEGAL DIRECT
D28D-D2BA  Perform DEF
D2BB-D2CD  Check FNx syntax
D2CE-D33E  evaluate FNx
D33F-D34E  Perform STR$
D34F-D360  Calculate string vector
D361-D3CD  Scan and set up string
D3CE-D3FF  Subroutine to build string vector
D400-D496  Garbage collection subroutine
D497-D4DF  Check for most eligible string collection
D4E0-D516  Collect a string
D517-D553  Perform string concatenation
D554-D57C  Build string into memory
D57D-D5B4  Discard unwanted string
D5B5-D5C5  Clean the descriptor stack
D506-D5D9  Perform CHR$
D5DA-D605  Perform LEFT$
D606-D610  Perform RIGHT$
D611-D63A  Perform MID$
D63B-D655  Pull string function parameters from stack
D656-D65B  Perform LEN
D65C-D664  Move from string-mode to numeric-mode
D665-D674  Perform ASC
D675-D686  Input byte parameter
D687-D605  Perform VAL
D6C6-D6D1  Get two parameters for POKE or WAIT
D6D2-D6E7  Convert floating point to fixed point
D6E8-D706  Perform PEEK
D707-D70F  Perform POKE
D710-D72B  Perform WAIT
D72C-D732  Add 0.5 to accumulator#1
D733-D744  Perform subtraction
D745-D76D  Microsoft joke
D76E-D852  Perform addition
D853-D889  Complement accumulator#1
D88A-D88E  Print OVERFLOW and exit
D88F-D8C7  Multiply-a-byte subroutine
D8C8-D8F5  Function constants: 1, SOR(.5),SOR(2), -00.5. etc.
D8F6-D936  Perform LOG
D937-D964  Perform multiplication
```

```
D965-D997  Multiply-a-bit subroutine
D998-D9C2  Load accumulator #2 from memory
D9C3-D9DF  Test and adjust accumulators #1 and #2
D9E0-D9ED  Handle overflow and underflow
D9EE-DA04  Multiply by 10
DA05-DA09  10 in floating binary
DA0A-DA12  Divide by 10
DA13-DA1D  Perform divide-into
DA1E-DAAD  Perform divide-by
DAAE-DAD2  Load accumulator #1 from memory
DAD3-DB07  Store accumulator #1 into memory
DB08-DB17  Copy accumulator #2 into accumulator #1
DB18-DB26  Copy accumulator #1 into accumulator #2
 DB27-DB36  Round off accumulator #1
DB37-DB44  Compute SGN value of accumulator #1
DB45-DB63  Perform SGN
DB64-DB66  Perform ABS
DB67-DBA6  Compare accumulator #1 to memory
DBA7-DBD7  Convert floating-point to-fixed-point
DBD8-DBFE  Perform INT
DBFF-DC89  Convert string to floating-point
DC8A-DCBE  Get new ASCII digit
DCBF-DCCD  String conversion constants: 99999999,999999999,  1E+9
DCCE-DCD8  Print IN, followed by:
DCD9-DCE8  Print Basic line number
DCE9-DE1C  Convert number or TI$ to ASCII
DE1D-DE5D  Constants for numeric conversion
DE5E-DE67  Perform SQR
DE68-DEA0  Perform power function
DEA1-DEAB  Perform negation
DEAC-DED9  Constants for string evaluation
DEDA-DF2C  Perform EXP
DF2D-DF76  Function series evaluation subroutines
DF77-DF7E  Manipulation constants for RND
DF7F-DFD7  Perform RND
DFD8-DFDE  Perform COS
DFDF-E027  Perform SIN
E028-E053  Perform TAN
E054-E08B  Constants for trig evaluation: pi/2, 2#pi, .25, etc.
E08C-E0BB  Perform ATN
```

E0BC-E0F8 Constants for ATN series evaluation
E0F9-E110 Subroutine to be moved to zero page ($70 to$87)
E111-E115 Initial RND seed
E116-E1B6 Initialize Basic system
E1B7-E1DD Messages: BYTES FREE, ### COMMODORE BASIC ###
E1DE-E228 Initialize I/O register, and:
E229-E256 Clear screen, and:
E257-E284 Home cursor
E285-E2F3 Input from screen or keyboard; wait for input completion
E2F4-E33E Input from screen
E33F-E34B Test for quotation mark and reverse quote-flag
E34C-E38A Set up screen print parameters
E38B-E395 Prevent 80-character line from getting any longer
E396-E3B3 Extend 40-character line to 80 characters
E3B4-E3D7 Back into the previous line (via DEL or CURSOR LEFT key)
E3D8-E518 Handle ASII character for screen output
E519-E53E Go to next line on screen
E53F-E5B9 Scroll the screen
E5BA-E61A Open a line on the screen (via INSERT key)
E61B-E62D Main interrupt entry point
E62E-E6E9 Hardware interrupt: service clock,keyboard,cassettes
E6EA-E6F7 Print character on screen
E6F8-E769 Table: decoder for keyboard matrix
E76A-E796 MLM subroutine: output hex digits
E797-E7A6 MLM subroutine: swap TMP0 and TMP2
E7A7-E7F6 MLM sibroutine: input hex digits
E7F7-E7FF MLM subroutine: print ?
F000-F0B5 Monitor messages, mostly for Input/Output
F0B6-F0ED Set up IEEE for Talk, Listen etc
F0EE-F127 Send character to IEEE-488 bus
F128-F135 Output character immediate mode to IEEE-488 bus
F136-F155 Send errors: WRITE TIMEOUT, DEVICE NOT PRESENT, etc
F156-F163 Send canned I/O message
F164-F16E Send immediate mode Listen command, then secondary address
F165-F17E Output character deferred mode to IEEE-488
F17F-F18B Drop IEEE channel: send Unlisten or Untalk
F18C-F1D0 Input character from IEEE-488 bus
F1D1-F1E0 GET a character
F1E1-F231 INPUT from any device
F232-F26D OUTPUT a character to any device

F26E-F283 Abort all files, and;
F284-F28C Restore normal I/O devices
F28D-F2A8 Find file table entry; set parameters from file table
F2A9-F300 Perform CLOSE
F301-F30E Test STOP key
F30F-F314 Action STOP key
F315-F31C Send message if direct mode
F31D-F321 Test if direct mode
F322-F3C1 Perform program loading
F3C2-F409 Perform LOAD
F40A-F43D Subroutines: Print SEARCHING...; Print LOADING or VERIFYING
F43E-F45F Get Load or Save parameters
F460-F465 Get a byte parameter
F466-F493 Send program name to IEEE-488 bus
F494-F4B6 Find a specific tape header
F4B7-F4CD Perform VERIFY
F4CE-F50D Get parameters for OPEN, CLOSE
F50E-F515 Abort calling subroutines if end-of-line (default parameters)
F516-F520 Confirm comma, else send SYNTAX ERROR
F521-F5A5 Perform OPEN
F5A6-F5D9 Find any tape header
F5DA-F63B Write tape header
F63C-F655 Get start and end program addresses from tape header
F656-F66B Set cassette buffer address according to device number
F66C-F683 Set tape start and end addresses from buffer address
F684-F68C Perform CMD
F68D-F69D Set tape start and end addresses from Basic pointers
F69E-F728 Perform SAVE
F729-F76C Update TI and TI$, and copy STOP key to work area
F76D-F76F TI constant: limit of clock (24 hours)
F770-F7BB Set input device
F7BC-F805 Set output device
F806-F811 Advance tape buffer pointer (for INPUT#, GET#, and PRINT#)
F812-F834 Wait: PRESS PLAY ON TAPE#
F835-F846 Test if cassette button(s) pressed
F847-F854 Wait: PRESS PLAY & RECORD ON TAPE#
F855-F885 Initiate tape read
F886-F8E5 Initiate tape write
F8E6-F8EF Test for I/O interrupt completion
F8F0-F8FF Test stop key

```
F900-F930  Set expected timing for next input bit from tape
F931-FA56  Interrupt entry: Read tape bits
FA57-FB75  Store received tape characters
FB76-FB7E  Set tape read/write address back to starting point
FB7F-FB83  Flag I/O error into ST
FB84-FB92  Reset 8-counter and flags for a new byte
FB93-FBAE  Write a transition to cassette tape
FBAF-FC40  Write interrupt 2: write data to tape
FC41-FC7A  Write interrupt 1: Write tape shorts (leader)
FC7B-FC95  Terminate tape: restore normal interrupt vector
FC96-FCA5  Set interrupt vector from table
FCA6-FCB3  Turn off cassette motors
FCB4-FCC5  Perform running checksum calculation
FCC6-FCD0  Check: read/write pointer at limit?
FCD1-FCFD  Power on reset entry point
FCFE-FD00  NMI interrupt entry point
FD01-FD10  Table of interrupt vectors
FD11-FFB0  Machine Language Monitor (MLM) - see Commodore documentation
FFB1-FFBF  Commodore copyright statement

*****JUMP TABLE*****
FFC0  OPEN
FFC3  CLOSE
FFC6  Set input device
FFC9  Set output device
FFCC  Restore default I/O devices
FFCF  Input character
FFD2  Output character
FFD5  LOAD
FFD8  SAVE
FFDB  VERIFY
FFDE  SYS
FFE1  Test STOP key
FFE4  Get character
FFE7  Abort all I/O activity
FFEA  Clock update
FFF0-FFF9  Unused
FFFA-FFFF  Hardware vectors: NMI, Reset, Interrupt
```

BASIC 4.0 MEMORY MAP

Compiled by Jim Butterfield

There are some differences between usage between the 40- and 80-column machines.

Hex	Decimal	Description
0000-0002	0-2	USR jump
0003	3	Search character
0004	4	Scan-between-quotes flag
0005	5	Input buffer pointer; # of subscripts
0006	6	Default DIM flag
0007	7	Type: FF=string, 00=numeric
0008	8	Type: 80=integer, 00=floating point
0009	9	Flag: DATA scan; LIST quote; memory
000A	10	Subscript flag; FNX flag
000B	11	0=INPUT; $40=GET; $98=READ
000C	12	ATN sign/Comparison Evaluation flag
000D-000F	13-15	Disk status DS$ descriptor
0010	16	Current I/O device for prompt-suppress
0011-0012	17-18	Integer value (for SYS, GOTO etc)
0013-0015	19-21	Pointers for descriptor stack
0016-001E	22-30	Descriptor stack(temp strings)
001F-0022	31-34	Utility pointer area
0023-0027	35-39	Product area for multiplication
0028-0029	40-41	Pointer: Start-of-Basic
002A-002B	42-43	Pointer: Start-of-Variables
002C-002D	44-45	Pointer: Start-of-Arrays
002E-002F	46-47	Pointer: End-of-Arrays
0030-0031	48-49	Pointer: String-storage(moving down)
0032-0033	50-51	Utility string pointer
0034-0035	52-53	Pointer: Limit-of-memory
0036-0037	54-55	Current Basic line number
0038-0039	56-57	Previous Basic line number
003A-003B	58-59	Pointer: Basic statement for CONT
003C-003D	60-61	Current DATA line number
003E-003F	62-63	Current DATA address
0040-0041	64-65	Input vector
0042-0043	66-67	Current variable name
0044-0045	68-69	Current variable address
0046-0047	70-71	Variable pointer for FOR/NEXT
0048-0049	72-73	Y-save; op-save; Basic pointer save
004A	74	Comparison symbol accumulator
004B-0050	75-80	Misc work area, pointers, etc
0051-0053	81-83	Jump vector for functions
0054-005D	84-93	Misc numeric work area
005E	94	Accum#1: Exponent
005F-0062	95-98	Accum#1: Mantissa
0063	99	Accum#1: Sign
0064	100	Series evaluation constant pointer
0065	101	Accum#1 hi-order (overflow)
0066-006B	102-107	Accum#2: Exponent, etc.
006C	108	Sign comparison, Acc#1 vs #2
006D	106	Accum#1 lo-order (rounding)
006E-006F	110-111	Cassette buff len/Series pointer
0070-0087	112-135	CHRGET subroutine; get Basic char
0077-0078	119-120	Basic pointer (within subrtn)
0088-008C	136-140	Random number seed.
008D-008F	141-143	Jiffy clock for TI and TI$
0090-0091	144-145	Hardware interrupt vector
0092-0093	146-147	BRK interrupt vector
0094-0095	148-149	NMI interrupt vector
0096	150	Status word ST
0097	151	Which key down; 255=no key
0098	152	Shift key: 1 if depressed
0099-009A	153-154	Correction clock
009B	155	Keyswitch PIA: STOP and RVS flags
009C	156	Timing constant for tape
009D	157	Load=0, Verify=1
009E	158	Number of characters in keybd buffer
009F	159	Screen reverse flag
00A0	160	IEEE output; 255=character pending
00A1	161	End-of-line-for-input pointer
00A3-00A4	163-164	Cursor log (row, column)
00A5	165	IEEE output buffer
00A6	166	Key image

00A7	167	0=flash cursor
00A8	168	Cursor timing countdown
00A9	169	Character under cursor
00AA	170	Cursor in blink phase
00AB	171	EOT received from tape
00AC	172	Input from screen/from keyboard
00AD	173	X save
00AE	174	How many open files
00AF	175	Input device, normally 0
00B0	176	Output CMD device, normally 3
00B1	177	Tape character parity
00B2	178	Byte received flag
00B3	179	Logical Address temporary save
00B4	180	Tape buffer character; MLM command
00B5	181	File name pointer; MLM flag, counter
00B7	183	Serial bit count
00B9	185	Cycle counter
00BA	186	Tape writer countdown
00BB-00BC	187-188	Tape buffer pointers, #1 and #2
00BD	189	Write leader count; read pass1/2
00BE	190	Write new byte; read error flag
00BF	191	Write start bit; read bit seq error
00C0-00C1	192-193	Error log pointers, pass1/2
00C2	194	0=Scan/1-15=Count/$40=Load/$80=End
00C3	195	Write leader length; read checksum
00C4-00C5	196-197	Pointer to screen line
00C6	198	Position of cursor on above line
00C7-00C8	199-200	Utility pointer: tape, scroll
00C9-00CA	201-202	Tape end addrs/End of current program
00CB-00CC	203-204	Tape timing constants
00CD	205	0=direct cursor, else programmed
00CE	206	Tape read timer 1 enabled
00CF	207	EOT received from tape
00D0	208	Read character error
00D1	209	# characters in file name
00D2	210	Current file logical address
00D3	211	Current file secondary addrs
00D4	212	Current file device number
00D5	213	Right-hand window or line margin
00D6-00D7	214-215	Pointer: Start of tape buffer
00D8	216	Line where cursor lives
00D9	217	Last key/checksum/misc.
00DA-00DB	218-219	File name pointer
00DC	220	Number of INSERTs outstanding
00DD	221	Write shift word/read character in
00DE	222	Tape blocks remaining to write/read
00DF	223	Serial word buffer
00E0-00F8	224-248	(40-column) Screen line wrap table
00E0-00E1	224-225	(80-column) Top, bottom of window
00E2	226	(80-column) Left window margin
00E3	227	(80-column) Limit of keybd buffer
00E4	228	(80-column) Key repeat flag
00E5	229	(80-column) Repeat countdown
00E6	230	(80-column) New key marker
00E7	231	(80-column) Chime time
00E8	232	(80-column) HOME count
00E9-00EA	233-234	(80-column) Input vector
00EB-00EC	235-236	(80-column) Output vector
00F9-00FA	249-250	Cassette status, #1 and #2
00FB-00FC	251-252	MLM pointer/Tape start address
00FD-00FE	253-254	MLM, DOS pointer, misc.
0100-010A	256-266	STR$ work area, MLM work
0100-013E	256-318	Tape read error log
0100-01FF	256-511	Processor stack
0200-0250	512-592	MLM work area; Input buffer
0251-025A	593-602	File logical address table
025B-0264	603-612	File device number table
0265-026E	613-622	File secondary adds table
026F-0278	623-632	Keyboard input buffer
027A-0339	634-825	Tape#1 input buffer
033A-03F9	826-1017	Tape#2 input buffer
033A	826	DOS character pointer
033B	827	DOS drive 1 flag
033C	828	DOS drive 2 flag
033D	829	DOS length/write flag
033E	830	DOS syntax flags

033F-0340	831-832	DOS disk ID
0341	833	DOS command string count
0342-0352	834-850	DOS file name buffer
0353-0380	851-896	DOS command string buffer
03EE-03F7	1006-1015	(80-column) Tab stop table
03FA-03FB	1018-1019	Monitor extension vector
03FC	1020	IEEE timeout defeat
0400-7FFF	1024-32767	Available RAM including expansion
8000-83FF	32768-33791	(40-column) Video RAM
8000-87FF	32768-34815	(80-column) Video RAM
9000-AFFF	36864-45055	Available ROM expansion area
B000-DFFF	45056-57343	Basic, DOS, Machine Lang Monitor
E000-E7FF	57344-59391	Screen, Keyboard, Interrupt programs
E810-E813	59408-59411	PIA 1 - Keyboard I/O
E820-E823	59424-59427	PIA 2 - IEEE-488 I/0
E840-E84F	59456-59471	VIA - I/O and timers
E880-E881	59520-59521	(80-column) CRT Controller
F000-FFFF	61440-65535	Reset, I/O handlers, Tape routines

PET 4.0 ROM ROUTINES

Jim Butterfield
Toronto

The 40-character and 80-character machines are the same except for addresses $E000-$E7FF.

This map shows where various routines lie. The first address is not necessarily the proper entry point for the routine. Similarly, many routines require register setup or data preparation before calling.

	Description
B000-B065	Action addresses for primary keywords
B066-B093	Action addresses for functions
B094-B0B1	Hierarchy and action addresses for operators
B0B2-B20C	Table of Basic keywords
B20D-B321	Basic messages, mostly error messages
B322-B34F	Search the stack for FOR or GOSUB activity
B350-B392	Open up space in memory
B393-B39F	Test: stack too deep?
B3A0-B3CC	Check available memory
B3CD	Send canned error message, then:
B3FF-B41E	Warm start; wait for Basic command
B41F-B4B5	Handle new Basic line input
B4B6-B4E1	Rebuild chaining of Basic lines
B4E2-B4FA	Receive line from keyboard
B4FB-B5A2	Crunch keywords into Basic tokens
B5A3-B5D1	Search Basic for given line number
B5D2	Perform NEW, and;
B5EC-B621	Perform CLR
B622-B62F	Reset Basic execution to start
B630-B6DD	Perform LIST
B6DE-B784	Perform FOR
B785-B7B6	Execute Basic statement
B7B7-B7C5	Perform RESTORE
B7C6-B7ED	Perform STOP or END
B7EE-B807	Perform CONT
B808-B812	Perform RUN
B813-B82F	Perform GOSUB
B830-B85C	Perform GOTO
B85D	Perform RETURN, then:
B883-B890	Perform DATA: skip statement
B891	Scan for next Basic statement
B894-B8B2	Scan for next Basic line
B8B3	Perform IF, and perhaps:
B8C6-B8D5	Perform REM: skip line
B8D6-B8F5	Perform ON
B8F6-B92F	Accept fixed-point number
B930-BA87	Perform LET
BA88-BA8D	Perform PRINT#
BA8E-BAA1	Perform CMD
BAA2-BB1C	Perform PRINT
BB1D-BB39	Print string from memory

BB3A-BB4B Print single format character
BB4C-BB79 Handle bad input data
BB7A-BBA3 Perform GET
BBA4-BBBD Perform INPUT#
BBBE-BBF4 Perform INPUT
BBF5-BC01 Prompt and receive input
BC02-BCF6 Perform READ
BCF7-BD18 Canned Input error messages
BD19-BD71 Perform NEXT
BD72-BD97 Check type mismatch
BD98 Evaluate expression
BEE9 Evaluate expression within parentheses
BEEF Check parenthesis, comma
BF00-BF0B Syntax error exit
BF8C-C046 Variable name setup
C047-C085 Set up function references
C086-C0B5 Perform OR, AND
C0B6-C11D Perform comparisons
C11E-C12A Perform DIM
C12B-C1BF Search for variable
C1C0-C2C7 Create new variable
C2C8-C2D8 Setup array pointer
C2D9-C2DC 32768 in floating binary
C2DD-C2FB Evaluate integer expression
C2FC-C4A7 Find or make array
C4A8 Perform FRE, and:
C4BC-C4C8 Convert fixed-to-floating
C4C9-C4CE Perform POS
C4CF-C4DB Check not Direct
C4DC-C509 Perform DEF
C50A-C51C Check FNx syntax
C51D-C58D Evaluate FNx
C58E-C59D Perform STR$
C59E-C5AF Do string vector
C5B0-C61C Scan, set up string
C61D-C669 Allocate space for string
C66A-C74E Garbage collection
C74F-C78B Concatenate
C78C-C7B4 Store string
C7B5-C810 Discard unwanted string
C811-C821 Clean descriptor stack
C822-C835 Perform CHR$
C836-C861 Perform LEFT$
C862-C86C Perform RIGHT$
C86D-C896 Perform MID$
C897-C8B1 Pull string data
C8B2-C8B7 Perform LEN
C8B8-C8C0 Switch string to numeric
C8C1-C8D0 Perform ASC
C8D1-C8E2 Get byte parameter
C8E3-C920 Perform VAL
C921-C92C Get two parameters for POKE or WAIT
C92D-C942 Convert floating-to-fixed
C943-C959 Perform PEEK
C95A-C962 Perform POKE
C963-C97E Perform WAIT
C97F-C985 Add 0.5
C986 Perform subtraction
C998-CA7C Perform addition
CA7D-CAB3 Complement accum#1
CAB4-CAB8 Overflow exit
CAB9-CAF1 Multiply-a-byte
CAF2-CB1F Constants
CB20 Perform LOG
CB5E-CBC1 Perform multiplication
CBC2-CBEC Unpack memory into accum#2
CBED-CC09 Test & adjust accumulators
CC0A-CC17 Handle overflow and underflow
CC18-CC2E Multiply by 10
CC2F-CC33 10 in floating binary
CC34 Divide by 10
CC3D Perform divide-by
CC45-CCD7 Perform divide-into
CCD8-CCFC Unpack memory into accum#1
CCFD-CD31 Pack accum#1 into memory
CD32-CD41 Move accum#2 to #1

```
CD42-CD50 Move accum#1 to #2
CD51-CD60 Round accum#1
CD61-CD6E Get accum#1 sign
CD6F-CD8D Perform SGN
CD8E-CD90 Perform ABS
CD91-CDD0 Compare accum#1 to memory
CDD1-CE01 Floating-to-fixed
CE02-CE28 Perform INT
CE29-CEB3 Convert string to floating-point
CEB4-CEE8 Get new ASCII digit
CEE9-CEF8 Constants
CF78      Print IN, then:
CF7F-CF92 Print Basic line #
CF93-D0C6 Convert floating-point to ASCII
D0C7-D107 Constants
D108      Perform SQR
D112      Perform power function
D14B-D155 Perform negation
D156-D183 Constants
D184-D1D6 Perform EXP
D1D7-D220 Series evaluation
D221-D228 RND constants
D229-D281 Perform RND
D282      Perform COS
D289-D2D1 Perform SIN
D2D2-D2FD Perform TAN
D2FE-D32B Constants
D32C-D35B Perform ATN
D35C-D398 Constants
D399-D3B5 CHRGET sub for zero page
D3B6-D471 Basic cold start
D472-D716 Machine Language Monitor
D717-D7AB MLM subroutines
D7AC-D802 Perform RECORD
D803-D837 Disk parameter checks
D838-D872 Dummy disk control messages
D873-D919 Perform CATALOG or DIRECTORY
D91A-D92E Output
D92F-D941 Find spare secondary address
D942-D976 Perform DOPEN
D977-D990 Perform APPEND
D991-D9D1 Get disk status
D9D2-DA06 Perform HEADER
DA07-DA30 Perform DCLOSE
DA31-DA64 Set up disk record
DA65-DA7D Perform COLLECT
DA7E-DAA6 Perform BACKUP
DAA7-DAC6 Perform COPY
DAC7-DAD3 Perform CONCAT
DAD4-DB0C Insert command string values
DB0D-DB39 Perform DSAVE
DB3A-DB65 Perform DLOAD
DB66-DB98 Perform SCRATCH
DB99-DB9D Check Direct command
DB9E-DBD6 Query ARE YOU SURE?
DBD7-DBE0 Print BAD DISK
DBE1-DBF9 Clear DS$ and ST
DBFA-DC67 Assemble disk command string
DC68-DE29 Parse Basic DOS command
DE2C-DE48 Get Device number
DE49-DE86 Get file name
DE87-DE9C Get small variable parameter
** Entry points only for E000-E7FF **
E000      Register/screen initialization
E0A7      Input from keyboard
E116      Input from screen
E202      Output character
E442      Main Interrupt entry
E455      Interrupt: clock, cursor, keyboard
E600      Exit from Interrupt
**                                  **
F000-F0D1 File messages
F0D2      Send 'Talk'
F0D5      Send 'Listen'
F0D7      Send IEEE command character
F109-F142 Send byte to IEEE
```

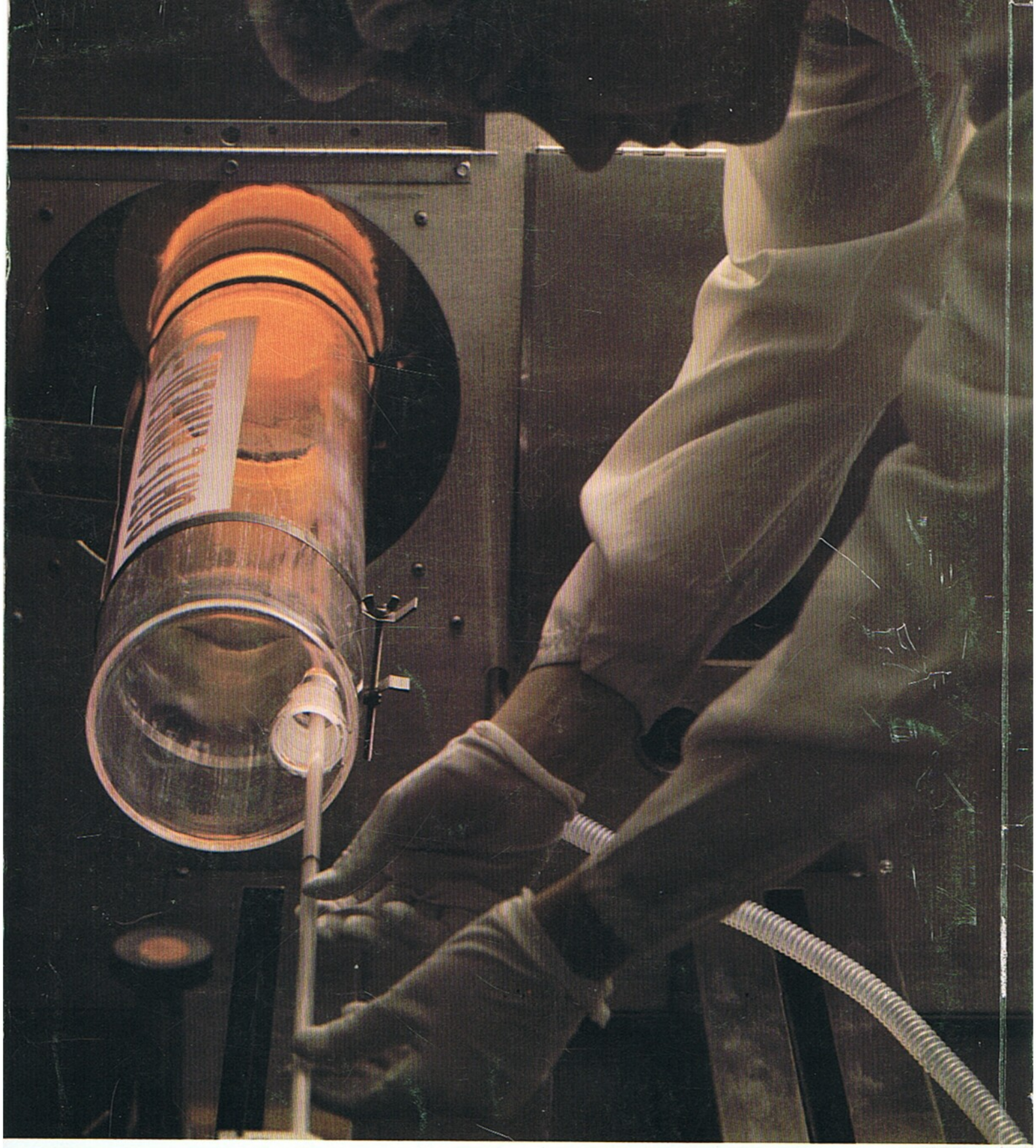
```
F143-F150 Send byte and clear ATN
F151-F16B Option: timeout or wait
F16C-F16F DEVICE NOT PRESENT
F170-F184 Timeout on read, clear control lines
F185-F192 Send canned file message
F193-F19D Send byte, clear control lines
F19E-F1AD Send normal (deferred) IEEE char
F1AE-F1BF Drop IEEE device
F1C0-F204 Input byte from IEEE
F205-F214 GET a byte
F215-F265 INPUT a byte
F266-F2A1 Output a byte
F2A2      Abort files
F2A6-F2C0 Restore default I/O devices
F2C1-F2DC Find/setup file data
F2DD-F334 Perform CLOSE
F335-F342 Test STOP key
F343-F348 Action STOP key
F349-F350 Send message if Direct mode
F351-F355 Test if Direct mode
F356-F400 Program load subroutine
F401-F448 Perform LOAD
F449-F46C Print SEARCHING
F46D-F47C Print LOADING or VERIFYING
F47D-F4A4 Get Load/Save parameters
F4A5-F4D2 Send name to IEEE
F4D3-F4F5 Find specific tape header
F4F6-F50C Perform VERIFY
F50D-F55F Get Open/Close parameters
F560-F5E4 Perform OPEN
F5E5-F618 Find any tape header
F619-F67A Write tape header
F67B-F694 Get start/end addrs from header
F695-F6AA Set buffer address
F6AB-F6C2 Set buffer start & end addrs
F6C3-F6CB Perform SYS
F6CC-F6DC Set tape write start & end
F6DD-F767 Perform SAVE
F768-F7AE Update clock
F7AF-F7FD Connect input device
F7FE-F84A Connect output device
F84B-F856 Bump tape buffer pointer
F857-F879 Wait for PLAY
F87A-F88B Test cassette switch
F88C-F899 Wait for RECORD
F89A      Initiate tape read
F8CB      Initiate tape write
F8E0-F92A Common tape I/O
F92B-F934 Test I/0 complete
F935-F944 Test STOP key
F945-F975 Tape bit timing adjust
F976-FA9B Read tape bits
FA9C-FBBA Read tape characters
FBBB-FBC3 Reset tape read address
FBC4-FBC8 Flag error into ST
FBC9-FBD7 Reset counters for new byte
FBD8-FBF3 Write a bit to tape
FBF4-FC85 Tape write
FC86-FCBF Write tape leader
FCC0-FCDA Terminate tape; restore interrupt
FCDB-FCEA Set interrupt vector
FCEB-FCF8 Turn off tape motor
FCF9-FD0A Checksum calculation
FD0B-FD15 Advance load/save pointer
FD16-FD4B Power-on Reset
FD4C-FD5C Table of interrupt vectors
** Jump table:                  **
FF93-FF9E CONCAT,DOPEN,DCLOSE,RECORD
FF9F-FFAA HEADER,COLLECT,BACKUP,COPY
FFAB-FFB6 APPEND,DSAVE,DLOAD,CATALOG
FFB7-FFBC RENAME,SCRATCH
FFBD      Get disk status
FFC0      OPEN
FFC3      CLOSE
FFC6      Set input device
FFC9      Set output device
```

```
FFCC        Restore default I/O devices
FFCF        INPUT a byte
FFD2        Output a byte
FFD5        LOAD
FFD8        SAVE
FFDB        VERIFY
FFDE        SYS
FFE1        Test stop key
FFE4        GET byte
FFE7        Abort all files
FFEA        Update clock
FFFA-FFFF   Hard vectors:  NMI, Reset, INT
```

Index

▲ Integrated Circuits being checked for their technical accuracy.

▼ Automatic IC and Component Insertion Equipment ensures high quality production of the main logic boards of the PET.



▲ Processing Silicon Wafers

Some of the subjects covered in this book:–

- Overlaying Programs
- Digital to Analogue Conversion
- LIFE – A machine code game for the PET
- Memory Maps for BASIC 1, 2 & 4 machines
- Sorting Techniques
- Direct Disk Access
- Learning Machine Code
- Book Reviews
- Applications

. . . plus much more!

commodore